
Parallel Programming Models Applicable to Cluster Computing and Beyond

Ricky A. Kendall,¹ Masha Sosonkina,¹ William D. Gropp,²
Robert W. Numrich,³ and Thomas Sterling⁴

¹ Scalable Computing Laboratory, Ames Laboratory, USDOE, Ames, IA 50011
{rickyk,masha}@scl.ameslab.gov

² Mathematics and Computer Science Division, Argonne National Laboratory,
Argonne, IL 60439 gropp@mcs.anl.gov

³ Supercomputing Institute, University of Minnesota, Minneapolis, MN 55455
rwn@msi.umn.edu

⁴ California Institute of Technology, Pasadena, CA 91125 tron@cacr.caltech.edu

Summary. This chapter centers mainly on successful programming models that map algorithms and simulations to computational resources used in high-performance computing. These resources range from group-based or departmental clusters to high-end resources available at the handful of supercomputer centers around the world. Also covered are newer programming models that may change the way we program high-performance parallel computers.

1 Introduction

Solving a system of partial differential equations (PDEs) lies at the heart of many scientific applications that model physical phenomena. The solution of PDES—often the most computationally intensive task of these applications—demands the full power of multiprocessor computer architectures combined with effective algorithms.

This synthesis is particularly critical for managing the computational complexity of the solution process when *nonlinear PDEs* are used to model a problem. In such a case, a mix of solution methods for large-scale nonlinear and linear systems of equations is used, in which a nonlinear solver acts as an “outer” solver. These methods may call for diverse implementations and programming models. Hence sophisticated software engineering techniques and a careful selection of parallel programming tools have a direct effect not only on the code reuse and ease of code handling but also on reaching the problem solution efficiently and reliably. In other words, these tools and techniques affect the numerical efficiency, robustness, and parallel performance of a solver.

For *linear PDEs*, the choice of a solution method may depend on the type of linear system of equations used. Many parallel direct and iterative

solvers are designed to solve a particular system type, such as symmetric positive definite linear systems. Many of the iterative solvers are also specific to the application and data format. There exists only a limited selection of “general-purpose” distributed-memory iterative-solution implementations. Among the better-known packages that contain such implementations are PETSc [2], *hypre* [11], and pARMS [51]. One common feature of these packages is that they are all based on domain decomposition methods and include a wide range of parallel solution techniques, such as preconditioners and accelerators.

Domain decomposition methods simply divide the domain of the problem into smaller parts and describe how solutions (or approximations to the solution) on each part is combined to give a solution (or approximation) to the original problem. For hyperbolic PDEs, these methods take advantage of the finite signal speed property. For elliptic, parabolic, and mixed PDEs, these methods take advantage of the fact that the influence of distant parts of the problem, while nonzero, is often small (for a specific example, consider the Green’s function for the solution to the Poisson problem). Domain decomposition methods have long been successful in solving PDEs on single processor computers (see, e.g., [71]), and lead to efficient implementations on massively parallel distributed-memory environments.⁵ Domain decomposition methods are attractive for parallel computing mainly because of their “divide-and-conquer” approach, to which many parallel programming models may be readily applied. For example, all three of the cited packages use the message-passing interface MPI for communication. When the complexity of the solution methods increases, however, the need to mix different parallel programming models or to look for novel ones becomes important. Such a situation may arise, for example, when developing a nontrivial parallel incomplete LU factorization, a direct sparse linear system solver, or any algorithm where data storage and movement are coupled and complex. The programming model(s) that provide(s) the best portability, performance, and ease of development or expression of the algorithm should be used. A good overview of applications, hardware and their interactions with programming models and software technologies is [17].

1.1 Programming Models

What is a programming model? In a nutshell it is the way one thinks about the flow and execution of the data manipulation for an application. It is an algorithmic mapping to a perceived architectural moiety.

In choosing a programming model, the developer must consider many factors: performance, portability, target architectures, ease of maintenance, code revision mechanisms, and so forth. Often, tradeoffs must be made among

⁵No memory is visible to all processors in a distributed-memory environment; each processor can only see their own local memory.

these factors. Trading computation for storage (either in memory or on disk) or for communication of data is a common algorithmic manipulation. The complexity of the tradeoffs is compounded by the use of parallel algorithms and hardware. Indeed, a programmer may have (as many libraries and applications do) multiple implementations of the same algorithm to allow for performance tuning on various architectures.

Today, many small and high-end high-performance computers are clusters with various communication interconnect technologies and with nodes⁶ having more than one processor. For example, the Earth Simulator [22] is a cluster of very powerful nodes with multiple vector processors; and large IBM SP installations (e.g., the system at the National Energy Research Scientific Computing Center, <http://hpcf.nersc.gov/computers/SP>) have multiple nodes with 4, 8, 16, or 32 processors each. These systems are at an abstract level the same kind of system. The fundamental issue for parallel computation on such clusters is how to select a programming model that gets the data in the right place when computational resources are available. This problem becomes more difficult as the number of processors increases; the term *scalability* is used to indicate the performance of an algorithm, method, or code, relative to a single processor. The scalability of an application is primarily the result of the algorithms encapsulated in the programming model used in the application. No programming model can overcome the scalability limitations inherent in the algorithm. There is no free lunch.

A generic view of a cluster architecture is shown in Figure 1. In the early Beowulf clusters, like the distributed-memory supercomputer shown in Figure 2, each node was typically a single processor. Today, each node in a cluster is usually at least a dual-processor symmetric processing (SMP) system. A generic view of an SMP node or a general shared-memory system is shown in Figure 3. The number of processors per computational node varies from one

Fig. 1. Generic architecture for a cluster system.

⁶A node is typically defined as a set of processors and memory that have a single system image; one operating system and all resources are visible to each other in the “node” moiety.

Fig. 2. Generic architecture for a distributed-memory cluster with a single processor.

installation to another. Often, each node is composed of identical hardware, with the same software infrastructure as well.

Fig. 3. Generic architecture for a shared-memory system.

The “view” of the target system is important to programmers designing parallel algorithms. Mapping algorithms with the chosen programming model to the system architecture requires forethought, not only about how the data is moved, but also about what type of hardware transport layer is used: for example, is data moved over a shared-memory bus between cooperating threads or over a fast Ethernet network between cooperating processes?

This chapter presents a brief overview of various programming models that work effectively on cluster computers and high-performance parallel supercomputers. We cannot cover all aspects of message-passing and shared-memory programming. Our goal is to give a taste of the programming models as well as the most important aspects of the models that one must consider in order to get an application parallelized. Each programming model takes a significant effort to master, and the learning experience is largely based on trial and error, with error usually being the better educational track. We also touch on newer techniques that are being used successfully and on a few

specialty languages that are gaining support from the vendor community. We give numerous references so that one can delve more deeply into any area of interest.

1.2 Application Development Efforts

“Best practices” for software engineering are commonly applied in industry but have not been so widely adopted in high-performance computing. Dubois outlines ten such practices for scientific programming [18]. We focus here on three of these.

The first is the use of a revision control system that allows multiple developers easy access to a central repository of the software. Both commercial and open source revision control systems exist. Some commonly used, freely available systems include Concurrent Versions System (CVS), Subversion, and BitKeeper. The functionality in these systems includes

- branching release software from the main development source,
- comparing modifications between versions of various subunits,
- merging modifications of the same subunit from multiple users, and
- obtaining a version of the development or branch software at a particular date and time.

The ability to recover previous instances of subunits of software can make debugging and maintenance easier and can be useful for speculative development efforts.

The second software engineering practice is the use of automatic build procedures. Having such procedures across a variety of platforms is useful in finding bugs that creep into code and inhibit portability. Automated identification of the language idiosyncrasies of different compilers minimizes efforts of porting to a new platform and compiler system. This is essentially normalizing the interaction of compilers and your software.

The third software engineering practice of interest is the use of a robust and exhaustive test suite. This can be coupled to the build infrastructure or, at a minimum, with every software release. The test suite should be used to verify the functionality of the software and, hence, the viability of a given release; it also provides a mechanism to ensure that ports to new computational resources are valid.

The cost of these software engineering mechanisms is not trivial, but they do make the maintenance and distribution easier. Consider the task of making Linux software distribution agnostic. Each distribution must have different versions of particular software moieties in addition to the modifications that each distribution makes to that software. Proper application of these tasks is essentially making one’s software operating system agnostic.

2 Message-Passing Interface

Parallel computing, with any programming model, involves two actions: *transferring data* among workers and *coordinating* the workers. A simple example is a room full of workers, each at a desk. The work can be described by written notes. Passing a note from one worker to another effects data transfer; receiving a note provides coordination (think of the note as requesting that the work described on the note be executed). This simple example is the background for the most common and most portable parallel computing model, known as *message passing*. In this section we briefly cover the message-passing model, focusing on the most common form of this model, the Message-Passing Interface (MPI).

2.1 The Message-Passing Interface

Message passing has a long history. Even before the invention of the modern digital computer, application scientists proposed halls full of skilled workers, each working on a small part of a larger problem and passing messages to their neighbors. This model of computation was formalized in computer science theory as communicating sequential processes (CSP) [35]. One of the earliest uses of message passing was for the Caltech Cosmic Cube, one of the first scalable parallel machines [70]. The success (perhaps more accurately, the potential success of highly parallel computing demonstrated by this machine) spawned many parallel machines, each with its own version of message passing.

In the early 1990s, the parallel computing market was divided among several companies, including Intel, IBM, Cray, Convex, Thinking Machines, and Meiko. No one system was dominant, and as a result the market for parallel software was splintered. To address the need for a single method for programming parallel computers, an informal group calling itself the MPI Forum and containing representatives from all stakeholders, including parallel computer vendors, applications developers, and parallel computing researchers, began meeting [32]. The result was a document describing a standard application programming interface (API) to the message-passing model, with bindings for the C and Fortran languages [53]. This standard quickly became a success. As is common in the development of standards, there were a few problems with the original MPI standard, and the MPI Forum released two updates, called MPI 1.1 and MPI 1.2. MPI 1.2 is the most widely available version today.

2.2 MPI 1.2

When MPI was standardized, most message-passing libraries at that time described communication between separate processes and contained three major components:

- Processing environment – information about the number of processes and other characteristics of the parallel environment.

- Point-to-point – messages from one process to another
- Collective – messages between a collection of processes (often all processes)

We will discuss each of these in turn. These components are the heart of the message passing programming model.

Processing Environment

In message passing, a parallel program comprises a number of separate processes that communicate by calling routines. The first task in an MPI program is to initialize the MPI library; this is accomplished with `MPI_Init`. When a program is done with MPI (usually just before exiting), it must call `MPI_Finalize`. Two other routines are used in almost all MPI programs. The first, `MPI_Comm_size`, returns in the second argument the number of processes available in the parallel job. The second, `MPI_Comm_rank`, returns in the second argument a ranking of the calling process, with a value between zero and size–1. Figure 4 shows a simple MPI program that prints the number of processes and the rank of each process. `MPI_COMM_WORLD` represents all the cooperating processes.

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    printf( "Hello World! I am %d of %d\n", rank, size );
    MPI_Finalize( );
    return 0;
}
```

Fig. 4. A simple MPI program.

While MPI did not specify a way to run MPI programs (much as neither C nor Fortran specifies how to run C or Fortran programs), most parallel computing systems require that parallel programs be run with a special program. For example, the program `mpixec` might be used to run an MPI program. Similarly, an MPI environment may provide commands to simplify compiling and linking MPI programs. For example, for some popular MPI implementations, the following steps will run the program in Figure 4 with four processes, assuming that program is stored in the file `first.c`:

```
mpicc -o first first.c
mpixec -n 4 first
```

The output may be

```
Hello World! I am 2 of 4
Hello World! I am 3 of 4
Hello World! I am 0 of 4
Hello World! I am 1 of 4
```

Note that the output of the process rank is not ordered from zero to three. MPI specifies that all routines that are *not* MPI routines behave independently, including I/O routines such as `printf`.

We emphasize that MPI describes communication between *processes*, not *processors*. For best performance, parallel programs are often designed to run with one process per processor (or, as we will see in the section on OpenMP, one thread per processor). MPI supports this model, but MPI also allows multiple processes to be run on a single-processor machine. Parallel programs are commonly developed on single-processor laptops, even with multiple processes. If there are more than a few processes per processor, however, the program may run very slowly because of contention among the processes for the resources of the processor.

Point-to-Point Communication

The program in Figure 4 is a very simple parallel program. The individual processes neither exchange data nor coordinate with each other. Point-to-point communication allows two processes to send data from one to another. Data is sent by using routines such as `MPI_Send` and is received by using routines such as `MPI_Recv` (we mention later several specialized forms for both sending and receiving).

We illustrate this type of communication in Figure 5 with a simple program that sums contributions from each process. In this program, each process first determines its rank and initializes the value that it will contribute to the sum. (In this case, the sum itself is easily computed analytically; this program is used for illustration only.) After receiving the contribution from the process with rank one higher, it adds the received value into its contribution and sends the new value to the process with rank one lower. The process with rank zero only receives data, and the process with the largest rank (equal to `size-1`) only sends data.

The program in Figure 5 introduces a number of new points. The most obvious are the two new MPI routines `MPI_Send` and `MPI_Recv`. These have similar arguments. Each routine uses the first three arguments to specify the data to be sent or received. The fourth argument specifies the destination (for `MPI_Send`) or source (for `MPI_Recv`) process, by rank. The fifth argument, called a *tag*, provides a way to include a single integer with the data; in this case the value is not needed, and a zero is used (the value used by the sender must match the value given by the receiver). The sixth argument specifies the collection of processes to which the value of rank is relative; we use

```

#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int          size, rank, valIn, valOut;
    MPI_Status status;

    MPI_Init( &argc, &argv );

    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* Pick a simple value to add */
    valIn = rank;

    /* receive the partial sum from the right processes (this is
       the sum from i=rank+1 to size-1) */
    if (rank < size - 1) {
        MPI_Recv( &valOut, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD,
                  &status );
        valIn += valOut;
    }
    /* Send the partial sum on to the left (rank-1) process */
    if (rank > 0) {
        MPI_Send( &valIn, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD );
    }
    else {
        printf( "The sum is %d\n", valOut );
    }

    MPI_Finalize( );
    return 0;
}

```

Fig. 5. A simple program to add values from each process.

MPI_COMM_WORLD, which is the collection of all processes in the parallel program (determined by the startup mechanism, such as `mpirun` in the “Hello World” example). There is one additional argument to `MPI_Recv`: `status`. This value contains some information about the message that some applications may need. In this example, we do not need the value, but we must still provide the argument.

The three arguments describing the data to be sent or received are, in order, the address of the data, the number of items, and the type of the data. Each basic datatype in the language has a corresponding MPI datatype, as shown in Table 1.

Table 1. Some major predefined MPI datatypes.

C		Fortran	
int	MPI_INT	INTEGER	MPI_INTEGER
float	MPI_FLOAT	REAL	MPI_REAL
double	MPI_DOUBLE	DOUBLE PRECISION	MPI_DOUBLE_PRECISION
char	MPI_CHAR	CHARACTER	MPI_CHARACTER
short	MPI_SHORT		

MPI allows the user to define new datatypes that can represent noncontiguous memory, such as rows of a Fortran array or elements indexed by an integer array (also called scatter-gathers). Details are beyond the scope of this chapter, however.

This program also illustrates an important feature of message-passing programs: because these are separate, communicating processes, all variables, such as `rank` or `valOut`, are private to each process and may (and often will) contain different values. That is, each process has its own memory space, and all variables are private to that process. The only way for one process to change or access data in another process is with the explicit use of MPI routines such as `MPI_Send` and `MPI_Recv`.

MPI provides a number of other ways in which to send and receive messages, including nonblocking (sometimes incorrectly called asynchronous) and synchronous routines. Other routines, such as `MPI_Iprobe`, can be used to determine whether a message is available for receipt. The nonblocking routines can be important in applications that have complex communication patterns and that send large messages. See [27, Chapter 4] for more details and examples.

Collective Communication and Computation

Any parallel algorithm can be expressed by using point-to-point communication. This flexibility comes at a cost, however. Unless carefully structured and documented, programs using point-to-point communication can be challenging to understand because the relationship between the part of the program that sends data and the part that receives the data may not be clear (note that well-written programs using point-to-point message passing strive to keep this relationship as plain and obvious as possible).

An alternative approach is to use communication that involves all processes (or all in a well-defined subset). MPI provides a wide variety of collective communication functions for this purpose. As an added benefit, these routines can be optimized for their particular operations (note, however, that these optimizations are often quite complex). As an example Figure 6 shows a program that performs the same computation as the program in Figure 5 but uses a single MPI routine. This routine, `MPI_Reduce`, performs a sum re-

duction (specified with `MPI_SUM`), leaving the result on the process with rank zero (the sixth argument).

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int          rank, valIn, valOut;
    MPI_Status status;

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* Pick a simple value to add */
    valIn = rank;

    /* Reduce to process zero by summing the values */
    MPI_Reduce( &valIn, &valOut, 1, MPI_INT, MPI_SUM, 0,
                MPI_COMM_WORLD );
    if (rank == 0) {
        printf( "The sum is %d\n", valOut );
    }

    MPI_Finalize( );
    return 0;
}
```

Fig. 6. Using collective communication and computation in MPI.

Note that this program contains only a single branch (if) statement that is used to ensure that only one process writes the result. The program is easier to read than its predecessor. In addition, it is effectively parallel; most MPI implementations will perform a sum reduction in time that is proportional to the log of the number of processes. The program in Figure 5, despite being a parallel program, will take time that is proportional to the number of processes because each process must wait for its neighbor to finish before it receives the data it needs to form the partial sum.⁷

Not all programs can be conveniently and efficiently written by using only collective communications. For example, for most MPI implementations, oper-

⁷One might object that the program in Figure 6 doesn't do exactly what the program in Figure 5 does because, in the latter, all of the intermediate results are computed and available to those processes. We offer two responses. First, only the value on the rank-zero process is printed; the others don't matter. Second, MPI offers the collective routine `MPI_Scan` to provide the partial sum results if that is required.

ations on PDE meshes are best done by using point-to-point communication, because the data exchanges are between pairs of processes and this closely matches the point-to-point programming model.

Other Features

MPI contains over 120 functions. In addition to nonblocking versions of point-to-point communication, there are routines for defining groups of processes, user-defined data representations, and testing for the availability of messages. These are described in any comprehensive reference on MPI [72, 27].

An important part of the MPI design is its support for programming in the large. Many parallel libraries have been written that make use of MPI; in fact, many applications can be written that have no explicit MPI calls and instead use libraries that themselves use MPI to express parallelism. Before writing any MPI program (or any program, for that matter), one should check to see whether someone has already done the hard work. See [28][Chapter 12] for a summary of some numerical libraries for Beowulf clusters.

2.3 The MPI-2 Extensions

The success of MPI created a desire to tackle some of the features not in the original MPI (henceforth called MPI-1). The major features include parallel I/O, the creation of new processes in the parallel program, and one-sided (as opposed to point-to-point) communication. Other important features include bindings for Fortran 90 and C++. The MPI-2 standard was officially released on July 18, 1997, and “MPI” now means the combined standard consisting of MPI-1.2 and MPI-2.0.

Parallel I/O

Perhaps the most requested feature for MPI-2 was parallel I/O. A major reason for using parallel I/O (as opposed to independent I/O) is performance. Experience with parallel programs using conventional file systems showed that many provided poor performance. Even worse, some of the most common file systems (such as NFS) are not designed to allow multiple processes to update the same file; in this case, data can be lost or corrupted. The goal for the MPI-2 interface to parallel I/O was to provide an interface that matched the needs of applications to create and access files in parallel, while preserving the flavor of MPI. This turned out to be easy. One can think of writing to a file as sending a message to the file system; reading a file is somewhat like receiving a message from the file system (“somewhat,” because one must ask the file system to send the data). Thus, it makes sense to use the same approach for describing the data to be read or written as is used for message passing—a tuple of address, count, and MPI datatype. Because the I/O is parallel, we

need to specify the group of processes; thus we also need a communicator. For performance reasons, we sometimes need a way to describe where the data is on the disk; fortunately, we can use MPI datatypes for this as well.

Figure 7 shows a simple program for reading a single integer value from a file. There are three steps, each similar to what one would use with non-parallel I/O:

1. Open the file. The `MPI_File_open` call takes a communicator (to specify the group of processes that will access the file), the file name, the access style (in this case, read-only), and another parameter used to pass additional data (usually empty, or `MPI_INFO_NULL`) and returns an `MPI_File` object that is used in MPI-IO calls.
2. Use all processes to read from the file. This simple call takes the file handle returned from `MPI_File_open`, the same buffer description (address, count, datatype) used in an `MPI_Recv` call, and (also like `MPI_Recv`) a status variable. In this case we use `MPI_STATUS_IGNORE` for simplicity.
3. Close the file.

```
/* Declarations, including */
MPI_File fh;
int val;

/* Start MPI */
MPI_Init( &argc, &argv );

/* Open the file for reading only */
MPI_File_open( MPI_COMM_WORLD, "input.dat", MPI_MODE_RDONLY,
               MPI_INFO_NULL, &fh );

/* All processes access the file and read the same value into
   val */
MPI_File_read_all( fh, &val, 1, MPI_INT, MPI_STATUS_IGNORE );
/* Close the file when no longer needed */
MPI_File_close( &fh );
```

Fig. 7. A simple program to read a single integer from a file.

Variations on this program, using other routines from MPI-IO, allow one to read different parts of the file to different processes and to specify from where in the file to read. As with message passing, there are also nonblocking versions of the I/O routines, with a special kind of nonblocking collective operation, called split-phase collective, available only for these I/O routines.

Writing files is similar to reading files. Figure 8 shows how each process can write the contents of the array `solution` with a single collective I/O call.

```

#define ARRAY_SIZE 1000
/* Declarations, including */
MPI_File fh;
int rank;
int solution[ARRAY_SIZE];

/* Start MPI */
MPI_Init( &argc, &argv );

/* Open the file for reading only */
MPI_File_open( MPI_COMM_WORLD, "output.dat", MPI_MODE_WRONLY,
               MPI_INFO_NULL, &fh );

/* Define where each process writes in the file */
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_File_set_view( fh, rank * ARRAY_SIZE * sizeof(double),
                  MPI_DOUBLE, MPI_DOUBLE, "native", MPI_INFO_NULL );
/* Perform the write */
MPI_File_write_all( fh, solution, ARRAY_SIZE, MPI_DOUBLE,
                   MPI_STATUS_IGNORE );
/* Close the file when no longer needed */
MPI_File_close( &fh );

```

Fig. 8. A simple program to write a distributed array to a file in a standard order that is independent of the number of processes.

Figure 8 illustrates the use of collective I/O, combined with *file views*, to efficiently write data from many processes to a single file in a way that provides a natural ordering for the data. Each process writes `ARRAY_SIZE` double-precision values to the file, ordered by the MPI rank of the process. Once this file is written, another program, using a different number of processes, can read the data in this file. For example, a non-parallel program could read this file, accessing all of the data.

Several good libraries provide convenient parallel I/O for user applications. Parallel netCDF [50] and HDF-5 [23] can read and write data files in a standard format, making it easy to move files between platforms. These libraries also encourage the inclusion of *metadata* in the file that describes the contents, such as the source of the computation and the meaning and units of measurements of the data. Parallel netCDF in particular encourages a collective I/O style for input and output, which helps ensure that the parallel I/O is efficient. We recommend that an I/O library be used if possible.

Dynamic Processes

Another feature that was often requested for MPI-2 was the ability to create and use additional processes. This is particularly valuable for ad hoc collections of desktop systems. Since MPI is designed for use on all kinds of parallel

computers, from collections of desktops to dedicated massively parallel computers, a scalable design was needed. MPI must also operate in a wide variety of environments, including ones where process creation is controlled by special process managers and schedulers.

In order to ensure scalability, process creation in MPI is collective, both over a group of processes that are creating new processes and over the group of processes created. The act of creating processes, or *spawning*, is accomplished with the routine `MPI_Comm_spawn`. This routine takes the name of the program to run, the command-line arguments for that program, the number of processes to create, the MPI communicator representing the group of processes that are spawning the new processes, a designated *root* (the rank of one process in the communicator that all members of that communicator agree to), and an `MPI_Info` object. The call returns a special kind of communicator, called an *intercommunicator*, that contains two groups of processes: the original group (from the input communicator) and the group of created processes. MPI point-to-point communication can then be used with this intercommunicator. The call also returns an array of error codes, one for each process.

Dynamic process creation is often used in master-worker programs, where the master process dynamically creates worker processes and then sends the workers tasks to perform. Such a program is sketched in Figure 9.

```

MPI_Comm workerIntercomm;
int errcodes[10];
...
MPI_Init( &argc, &argv );
...
MPI_Comm_spawn( "./worker", MPI_ARGV_NULL, 10, MPI_INFO_NULL, 0,
                MPI_COMM_SELF, &workerIntercomm, errcodes );

for (i=0; i<10; i++) {
    MPI_Send( &task, 1, MPI_INT, i, 0, workerIntercomm );
    ...
}

```

Fig. 9. Sketch of an MPI master program that creates 10 worker processes and sends them each a task, specified by a single integer.

MPI also provides routines to spawn different programs on different processes with `MPI_Comm_spawn_multiple`. Special values used for the `MPI_Info` parameter allow one to specify special requirements about the processes, such as their working directory.

In some cases two parallel programs may need to connect to each other. A common example is a climate simulation, where separate programs perform the atmospheric and ocean modeling. However, these programs need to share

data at the ocean-atmosphere boundary. MPI allows programs to connect to one another by using the routines `MPI_Comm_connect` and `MPI_Comm_accept`. See [29, Chapter 7] for more information.

One-Sided Communication

The message-passing programming model relies on the sender and receiver cooperating in moving data from one process to another. This model has many strengths but can be awkward, particularly when it is difficult to coordinate the sender and receiver. A different programming model relies on one-sided operations, where one process specifies both the source and the destination of the data moved between processes. Experience with BSP [34] and the Cray SHMEM [14] demonstrated the value of one-sided communication. The challenge for the MPI Forum was to design an interface for one-sided communication that retained the “look and feel” of MPI and could deliver good and reliable performance on a wide variety of platforms, including very fast computers without cache-coherent memory. The result was a compromise, but one that has been used effectively on one of the fastest machines in the world, the Earth Simulator.

In one-sided communication, a process may either *put* data into another process or *get* data from another process. The process performing the operation is called the *origin* process; the other process is the *target* process. The data movement happens without *explicit* cooperation between the origin and target processes. The origin process specifies both the source and destination of the data. A third operation, *accumulate*, allows the origin process to perform some basic operations, such as sum, with data at the target process. The one-sided model is sometimes called a put-get programming model.

Figure 10 sketches the use of `MPI_Put` for updating “ghost points” used in a one-dimensional finite difference grid. This has three parts:

1. One-sided operations may target only memory that has been marked as available for use by a particular memory window. The memory window is the one-sided analogue to the MPI communicator and ensures that only memory that the target process specifies may be updated by another process using MPI one-sided operations. The definition is made with the `MPI_Win_create` routine.
2. Data is moved by using the `MPI_Put` routine. The arguments to this routine are the data to put from the origin process (three arguments: address, count, and datatype), the rank of the target process, the destination of the data relative to the target window (three arguments: offset, count, and datatype), and the memory window object. Note that the destination is specified as an offset into the memory that the target process specified by using `MPI_Win_create`, not a memory address. This provides better modularity as well as working with heterogeneous collections of systems.
3. Because only the origin processes call `MPI_Put`, the target process needs some way to know when the data is available. This is accomplished with

the `MPI_Win_fence` routine, which is collective over all the processes that created the memory window (in this example, all processes). In fact, in MPI the put, get, and accumulate calls are all nonblocking (for maximum performance), and the `MPI_Win_fence` call ensures that these calls have completed at the origin processes.

```
#   define ARRAYSIZE
double x[ARRAYSIZE+2];
MPI_Win win;
int rank, size, leftNeighbor, rightNeighbor;

MPI_Init( &argc, &argv );
...
/* compute the neighbors.  MPI_PROC_NULL mean "no neighbor" */
leftNeighbor = rightNeighbor = MPI_PROC_NULL;
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (rank > 0) leftNeighbor = rank - 1;
if (rank < size - 1) rightNeighbor = rank + 1;

...
/* x[0] and x[ARRAYSIZE+1] are the ghost cells */
MPI_Win_create( x, (ARRAYSIZE+2) * sizeof(double), sizeof(double),
               MPI_INFO_NULL, MPI_COMM_WORLD, &win );

MPI_Win_fence( 0, win );
MPI_Put( &x[1], 1, MPI_DOUBLE,
        leftNeighbor, ARRAYSIZE+1, 1, MPI_DOUBLE, win );
MPI_Put( &x[ARRAYSIZE], 1, MPI_DOUBLE,
        rightNeighbor, 0, 1, MPI_DOUBLE, win );
MPI_Win_fence( 0, win );

...
MPI_Win_free( &win );
```

Fig. 10. Sketch of a program that uses MPI one-sided operations to communicate ghost cell data to neighboring processes.

While the MPI one-sided model is similar to other one-sided models, it has important differences. In particular, some models assume that the addresses of variables (particularly arrays) are the same on all processes. This assumption simplifies many features of the implementation and is true for many applications. MPI, however, does not assume that all programs are the same or that all runtime images are the same (e.g., running on heterogeneous platforms, which could be all IA32 processors but with different installed runtime libraries for C or Fortran). Thus, the address of `MyArray` in the program

on one processor may not be the same as the address of the variable with the same name on another processor (some programming models, such as Co-Array Fortran, do make and require this assumption; see Section 5.2).

While we have touched on the issue of synchronization, this is a deep subject and is reflected in the MPI standard. Reading the standard can create the impression that the MPI model is very complex, and in some ways this is correct. However, the complexity is designed to allow implementors the greatest flexibility while delivering precisely defined behavior. A few simple rules will guarantee the kind of behavior that many users expect and use. The full rules are necessary only when trying to squeeze the last bits of performance from certain kinds of computing platforms, particularly machines without fully cache-coherent memory systems, such as certain vector machines that are among the world's fastest. In fact, rules of similar complexity apply to shared-memory programming and are related to the pragmatic issues of memory consistency and tradeoffs between performance and simplicity.

Other Features in MPI-2

Among the most important other features in MPI-2 are bindings for C++ and Fortran 90. The C++ binding provides a low-level interface that exploits the natural objects in MPI. The Fortran 90 binding includes an MPI module, providing some argument checking for Fortran programs. Other features include routines to specify levels of thread safety and to support tools that must work with MPI programs. More information may be found in [26].

2.4 State of the Art

MPI is now over twelve years old. Implementations of MPI-1 are widespread and mature; many tools and applications use MPI on machines ranging from laptops to the world's largest and fastest computers. See [55] for a sampling of papers on MPI applications and implementations. Improvements continue to be made in the areas of performance, robustness, and new hardware. In addition, the parallel I/O part of MPI-2 is widely available.

Shortly after the MPI-2 standard was released, Fujitsu had an implementation of all of MPI-2 except for `MPI_Comm_join` and a few special cases of `MPI_Comm_spawn`. Other implementations, free or commercially supported, are now available for a wide variety of systems.

The MPI one-sided operations are less mature. Many implementations now support at least the "active target" model (these correspond to the BSP or put-get followed by barrier). In some cases, while the implementation of these operations is correct, the performance may not be as good as MPI's point-to-point operations. Other implementations have achieved good results, even on clusters with no special hardware to support one-sided operations [74]. Recent work exploiting the abilities of emerging network standards such as Infiniband shows how the MPI one-sided operations can provide excellent performance [41].

2.5 Summary

MPI provides a mature, capable, and efficient programming model for parallel computation. A large number of applications, libraries, and tools are available that make use of MPI. MPI applications can be developed on a laptop or desktop, tested on an ad hoc cluster of workstations or PCs, and then run in production on the world's largest parallel computers. Because MPI was designed to support "programming in the large," many libraries written with MPI are available, simplifying the task of building many parallel programs. MPI is also general and flexible; any parallel algorithm can be expressed in MPI. These and other reasons for the success of MPI are discussed in more detail in [30].

3 Shared-Memory Programming with OpenMP

Shared-memory programming on multiprocessor systems has been around for a long time. The typical generic architectural schematic for a shared-memory system or an individual SMP node in a distributed-memory system is shown in Figure 3. The memory of the system is directly accessible by all processors, but that access may be coupled by different bandwidth and latency mechanisms. The latter situation is often referred to as non-uniform memory access (NUMA). For optimal performance, parallel algorithms must take this into account.

The vendor community offers a huge number of shared-memory-based hardware systems, ranging from dual-processor systems to very large (e.g., 512-processor) systems. Many clusters are built from these shared-memory nodes, with two or four processors being common and a few now using 8-way systems. The relatively new AMD Opteron systems will be generally available in 8-way configurations within the coming year. More integrated parallel supercomputer systems such as the IBM SP have 16- or 32-way nodes.

Programming in shared memory can be done in a number of ways, some based on threads, others on processes. The main difference, by default, is that threads share the same process construct and memory, whereas multiple processes do not share memory. Message passing is a multiple process based programming model. Overall, thread-based models have some advantages. Creating an additional thread of execution is usually faster than creating another process, and synchronization and context switches among threads are faster than among processes. Shared-memory programming is in general incremental; a given section of code can be parallelized without modifying external data storage or data access mechanisms.

Many vendors have their own shared-memory programming models. Most offer System V interprocess communication (IPC) mechanisms, which include shared-memory segments and semaphores [31]. System V IPC usually shares memory segments among different processes. The Posix standard [40, 57] offers

a specific threads model called Pthreads. It has a generic interface that makes it more suitable for systems-level programming than for high-performance computing applications. Only one compiler (as far as we know) supports the Fortran Pthreads standard; C/C++ support is commonplace in Unix; and there is a one-to-one mapping of the Pthreads API to the Windows threads API as well, so the latter is a common shared-memory programming model available to the development community. Java threads also provides a mechanism for shared-memory concurrent programming [39].

Many other thread-based programming libraries are available from the research community as well, for example, TreadMarks [43]. These libraries are supported across a wide variety of platforms principally by the library development teams. OpenMP, on the other hand, is a shared-memory, thread-based programming model or API supported by the vendor community. Most commercial compilers available for Linux provide OpenMP support.

Overall, thread-based models have some advantages. Creating an additional thread of execution is usually faster than creating another process. Synchronization and context switches among threads are faster than among processes.

In the remainder of this section, we focus on the OpenMP programming model.

3.1 OpenMP History

OpenMP [12, 15] was organized in 1997 by the OpenMP Architecture Review Board (ARB), which owns the copyright on the specifications and manages the standard development. The ARB is composed primarily of representatives from the vendor community; membership is open to corporate, research, or academic institutions, not to individuals [64]. The goal of the original effort was to provide a shared-memory programming standard that combined the best practices of the vendor community offerings and some specifications that were a part of previous standardization efforts of the Parallel Computing Forum [49, 24] and the ANSI X3H5 [75] committee.

The ARB keeps the standard relevant by expanding the standard to meet needs and requirements of the user and development communities. The ARB also works to increase the impact of OpenMP and interprets the standard for the community as questions arise. The currently available version 2 standards for C/C++ [5] and Fortran [4] can be downloaded from the OpenMP ARB Web site [64]. The ARB has combined these standards into one working specification (version 2.5) for all languages, clarifying previous inconsistencies and strengthening the overall standard. The merged draft was released in November, 2004.

3.2 The OpenMP Model

OpenMP uses an execution model of fork and join (see Figure 11) in which the “master” thread executes sequentially until it reaches instructions that

essentially ask the runtime system for additional threads to do concurrent work. Once the concurrent scope of execution has completed, these extra threads simply go away, and the master thread continues execution serially. The details of the underlying threads of execution are compiler dependent and system dependent. In fact, some OpenMP implementations are developed on top of Pthreads. OpenMP uses a set of compiler directives, environment variables, and library functions to construct parallel algorithms within an application code. OpenMP is relatively easy to use and affords the ability to do incremental parallelism within an existing software package.

Fig. 11. Fork-and-join model of executing threads.

OpenMP uses a variety of mechanisms to construct parallel algorithms within an application code. These are a set of compiler directives, environment variables, and library functions. OpenMP is essentially an implicit parallelization method that works with standard C/C++ or Fortran. Various mechanisms are available for dividing work among executing threads, ranging from automatic parallelism provided by some compiler infrastructures to the ability to explicitly schedule work based on the thread ID of the executing threads. Library calls provide mechanisms to determine the thread ID and number of participating threads in the current scope of execution. There are also mechanisms to execute code on a single thread atomically in order to protect execution of critical sections of code. The final application becomes a series of sequential and parallel regions, for instance connected segments of the single serial-parallel-serial segment as shown in Figure 12.

Using OpenMP in essence involves three basic parallel constructs:

1. Expression of the algorithmic parallelism or controlling the flow of the code
2. Constructs for sharing data among threads or the specific communication mechanism involved
3. Synchronization constructs for coordinating the interactions among threads

These three basic constructs, in their functional scope, are similar to those used in MPI or any other parallel programming model.

Fig. 12. An OpenMP application using the fork-and-join model of executing threads has multiple concurrent teams of threads.

OpenMP directives are used to define blocks of code that can be executed in parallel. The blocks of code are defined by the formal block structure in C/C++ and by comments in Fortran; both the beginning and end of the block of code must be identified. There are three kinds of OpenMP directives: parallel constructs, work-sharing constructs within a parallel construct, and combined parallel-work-sharing constructs.

Communication is done entirely in the shared-memory space of the process containing threads. Each thread has a unique stack pointer and program counter to control execution in that thread. By default, all variables are shared among threads in the scope of the process containing the threads. Variables in each thread are either shared or private. Special variables, such as reduction variables, have both a shared scope and a private scope that changes at the boundaries of a parallel region. Synchronization constructs include mutual exclusions that control access to shared variables or specific functionality (e.g., regions of code). There are also explicit and implied barriers, the latter being one of the subtleties of OpenMP. In parallel algorithms, there must be a communication of critical information among the concurrent execution entities (threads or processes). In OpenMP, nearly all of this communication is handled by the compiler. For example, a parallel algorithm has to know the number of entities participating in the concurrent execution and how to identify the appropriate portion of the entire computation for each entity. This maps directly to a process-count- and process-identifier-based algorithm in MPI.

A simple example is in order to whet the appetite for the details to come. In the code segments in Figure 13 we have a “Hello World”-like program that uses OpenMP. This generic program uses a simple parallel region that

C code	Fortran Code
<code>#include <stdio.h></code>	<code>program hello</code>
<code>#include <omp.h></code>	<code>implicit none</code>
<code>int main(int argc, char *argv[])</code>	<code>integer tid</code>
<code>{</code>	<code>integer omp_get_thread_num</code>
<code> int tid;</code>	<code>external omp_get_thread_num</code>
<code>#pragma omp parallel private(tid)</code>	<code>!\$omp parallel private(tid)</code>
<code> {</code>	<code> tid = omp_get_thread_num()</code>
<code> tid = omp_get_thread_num();</code>	<code> write(6, '(1x,a1,i4,a1)')</code>
<code> printf("<%d>\n",tid);</code>	<code> & ' <',tid,' >'</code>
<code> }</code>	<code>!\$omp end parallel</code>
<code>}</code>	<code>end</code>

Fig. 13. “Hello World” OpenMP code.

designates the block of code to be executed by all threads. The C code uses the language standard braces to identify the block; the Fortran code uses comments to identify the beginning and end of the parallel region. In both

codes the OpenMP library function `omp_get_thread_num` returns the thread number, or ID, of the calling thread; the result is an integer value ranging from 0 to the number of threads minus 1. Note that type information for the OpenMP library function does not follow the default variable type scoping in Fortran. To run this program, one would execute the binary like any other binary. To control the number of threads used, one would set the environment variable `OMP_NUM_THREADS` to the desired value. What output should be expected from this code? Table 2 shows the results of five runs with the number of threads set to 3. The output from this simple example illustrates

Table 2. Multiple runs of the OpenMP “Hello World” program. Each column represents the output of a single run of the application on 3 threads.

Run 1	Run 2	Run 3	Run 4	Run 5
<0>	<2>	<1>	<0>	<0>
<1>	<1>	<0>	<1>	<1>
<2>	<0>	<2>	<2>	<2>

an important point about thread-based parallel programs, in OpenMP or any other thread model: There is no control over which thread executes first within the context of a parallel region. This decision is determined by the runtime system. Any expectation or required ordering of the execution of threads must be explicitly coded. The simple concurrency afforded by OpenMP requires that each task, such as a single iteration of a loop, be an independent execution construct.

One of the advantages of OpenMP is *incremental parallelization*—the ability to parallelize loops at a time or even small segments of code at a time. By iteratively identifying the most time-consuming components of an application and then parallelizing those components, one eventually gets a fully parallelized application. Any programming model requires a significant amount of testing and code restructuring to get optimal performance.⁸ Although the mechanisms of OpenMP are straightforward and easier than other parallel programming models, the cycle of restructuring and testing is still important. The programmer may introduce a bug by incorrectly parallelizing a code and introducing a dependency that goes undetected because the code was not then thoroughly tested. One should remember that the OpenMP user has no control on the order of thread execution; a few tests may detect a dependency—or may not. In other words the tests you run may just get “lucky” and give the correct results. We discuss dependency analysis further in Section 3.4.

⁸Some parallel software developers call parallelizing a code re-bugging a code, and this is often an apropos statement.

3.3 OpenMP Directives

The mechanics of parallelization with OpenMP are relatively straightforward. The first step is to insert compiler directives into the source code identifying the code segments or loops to be parallelized. Table 3 shows the sentinel syntax of a general directive for OpenMP in the supported languages [5, 4]. The

Table 3. General sentinel syntax of OpenMP directives.

Language	Syntax
Fortran 77	*\$omp directive [options] C\$omp directive [options] !\$omp directive [options]
Fortran 90/95	!\$omp directive [options]
Continuation Syntax	!\$omp directive [options] !\$omp+ directive [options]
C or C++	#pragma omp directive [options]
Continuation Syntax	#pragma omp directive [options] \ directive [options]

easiest way to learn how to develop OpenMP applications is through examples. We start with a simple algorithm, computing the norm of the difference of two vectors. This is a common way to compare vectors or matrices that are supposed to be the same. The serial code fragment in C and Fortran is shown in Figure 14. This simple example exposes some of the concepts needed to

<pre> C code fragment norm = (double) 0.0; for(i=0;i<len;i++) { diff = z[i]-zp[i]; norm += diff*diff; } </pre>	<pre> Fortran code fragment norm = 0.0d00 do i = 1,len diff = z(i) - zp(i) norm = norm + diff*diff enddo </pre>
---	---

Fig. 14. “Norm of vector difference” serial code.

appropriately parallelize a loop with OpenMP. By thinking about executing each iteration of the loop independently, we can see several issues with respect to reading from and writing to memory locations. First, we have to understand that each iteration of the loop essentially needs a separate **diff** memory location. Since **diff** for *each* iteration is *unique* and different iterations are

being executed concurrently on multiple threads, `diff` cannot be shared. Second, with all threads writing to `norm`, we have to ensure that all values are appropriately added to the memory location. This process can be handled in two ways: We can protect the summation into `norm` by a critical section (an atomic operation), or we can use a reduction clause to sum a thread local version of `norm` into the final value of `norm` in the master thread. Third, all threads of execution have to read the values of the vectors involved and the length of the vectors.

Now that we understand the “data” movement in the loop, we can apply directives to make the movement appropriate. Figure 15 contains the parallelized code using OpenMP with a critical section. We have identified `i` as private so that only one thread will execute a given value of `i`; each iteration is executed only once. Also private is `diff` because each thread of execution must have a specific memory location to store the difference; if `diff` were not private, the overlapped execution of multiple threads would not guarantee the appropriate value when it is read in the norm summation step. The “atomic” directive allows only one thread at a time to do the summation of `norm`, thereby ensuring that the correct values are summed into the shared variable. This is important because summation involves the data load, register operations, and data store. If this were not protected, multiple threads could overlap these operations. For example, thread 1 could load a value of `norm`, thread 2 could store an updated value of `norm`, and then thread 1 would have the wrong value of `norm` for the summation.

C code fragment

```
norm = (double) 0.0;
#pragma omp parallel for private(i,diff) shared(len,z,zp,norm)
  for(i=0;i<len;i++) {
    diff = z[i]-zp[i];
#pragma omp atomic
    norm += diff*diff;
  }
```

Fortran code fragment

```
norm = 0.0d00
!$OMP PARALLEL DO PRIVATE(i,diff) SHARED(len,z,zp,norm)
  do i = 1,len
    diff = z(i) - zp(i)
!$OMP ATOMIC
    norm = norm + diff*diff
  enddo
!$OMP END PARALLEL DO
```

Fig. 15. “Norm of vector difference” OpenMP code with a critical section.

Since all the threads have to execute the norm summation line atomically, there clearly will be contention for access to update the value of `norm`. This overhead, waiting in line to update the value, will severely limit the overall performance and scalability of the parallel loop.⁹ A better approach would be to have each thread sum into a private variable and then use the partial sums in each thread to compute the total `norm` value. This is what is done with a reduction clause. The variable in a reduction clause is private during the execution of the concurrent threads, and the value in each thread is reduced over the given operation and returned to the master thread just as a shared variable operates. This dual nature provides a mechanism to parallelize the algorithm without the need for the atomic operation as in Figure 16. This eliminates the thread contention of the atomic operation.

```

C code fragment
norm = (double) 0.0;
#pragma omp parallel for private(i,diff) shared(len,z,zp,norm) \
                        reduction(+:norm)
for(i=0;i<len;i++) {
    diff = z[i]-zp[i];
    norm += diff*diff;
}

Fortran code fragment
norm = 0.0d00
!$OMP PARALLEL DO PRIVATE(i,diff) SHARED(len,z,zp,norm)
!$OMP+      REDUCTION(+:norm)
do i = 1,len
    diff = z(i) - zp(i)
    norm = norm + diff*diff
enddo
!$OMP END PARALLEL DO

```

Fig. 16. “Norm of vector difference” OpenMP code with a reduction.

The reduction mechanism is a useful technique, and another example of the use of the reduction clause is in order. In developing parallel algorithms, one often measures their performance by timing the event in each execution entity, either in each thread or in each process. Knowing the minimum, maximum, and average time of concurrent tasks will give some indication of the level of load balance in the algorithm. If the minimum, maximum, and average times are all about the same, then the algorithm has good load-balance. If the minimum, maximum, or both are far away from the average then there is a load imbalance that has to be mitigated. This can be accomplished by some

⁹In fact, this simple example will not scale well regardless of the OpenMP mechanism used because the amount of work in each thread compared to the overhead of the parallelization is small.

sort of regrouping of elements of each task or via some dynamic mechanism. As a specific example, we will show code fragment for a sparse matrix vector multiplication in Figure 17. The sparse matrix is stored in the compressed-row-storage (CRS) format, a standard format that many sparse codes use in their algorithms. See [68] for details of various sparse matrix formats.

```

!      compute yvec = Amat*xvec
!      Amat sparse matrix stored in CRS format
!      Flat linear storage of elements of A
!      row_ptr() points to the start and of each row of A
!      in the flat linear storage of A. The last element
!      has the number of non-zero elements of A + 1.
!      Therefore each row has row_ptr(i+1)-row_ptr(i)-1
!      elements
!      col_ind() provides the column index for each element of A
!
do i = 1,n
!
!      compute the inner product of row i with vector xvec
!
      t = 0.0d0
      do k=row_ptr(i), row_ptr(i+1)-1
        t = t + amat(k)*xvec(col_ind(k))
      enddo
!
!      store result in yvec(i)
!
      yvec(i) = t
    enddo

```

Fig. 17. Sequential sparse matrix multiply code fragment, in Fortran.

To parallelize this loop using OpenMP, we have to determine the data flow in the algorithm. We will parallelize this code over the outer loop, *i*. Each iteration of that loop will be executed only once across all threads in the team. Each iteration is independent, so writing to *yvec(i)* is independent in each iteration. Therefore, we do not have to protect that write with an atomic directive as we did in the “norm” computation example. Hence, *yvec* needs to be shared because each thread will write to some part of the vector. The temporary summation variable *t* and the inner do loop variable *k* are different for each iteration of *i*. Thus, they must be private; that is, each thread must have a separate memory location. All other variables are only being read, so these variables are shared because all threads have to know all the values.

Figure 18 shows the parallelized code fragment. Timing mechanisms are inserted for the do loop and the reduction clause is inserted for each of the reduction variables, *timemin*, *timemax*, and *timeave*. The OpenMP library func-

```

!      compute yvec = Amat*xvec
...
!$OMP PARALLEL REGION PRIVATE(i,t,k,timestart,timeend,numthread)
!$OMP+      SHARED(n,row_ptr,amat,xvec,col_ind,yvec)
!$OMP+      REDUCTION(MIN:timemin) REDUCTION(MAX:timemax)
!$OMP+      REDUCTION(+:timeave)
      timestart = omp_get_wtime()
!$OMP PARALLEL DO
      do i = 1,n
        t = 0.0d0 ! inner product of row i with vector xvec
        do k=row_ptr(i), row_ptr(i+1)-1
          t = t + amat(k)*xvec(col_ind(k))
        enddo
        yvec(i) = t !      store result in yvec(i)
      enddo
!$OMP END PARALLEL DO
      timeend = omp_get_wtime()
      numthread = omp_get_num_threads()
      timemin = timeend-timestart
      timemax = timeend-timestart
      timeave = (timeend-timestart)/numthread
!$OMP END PARALLEL REGION

```

Fig. 18. Parallel sparse matrix multiply code fragment, in Fortran, that times the operation and reduces the minimum, maximum, and average times.

tion `omp_get_wtime()` returns a double-precision clock tick based on some implementation dependent epoch. The library function `omp_get_num_threads()` returns the total number of threads in the team of the parallel region. The defaults are used for scheduling the iterations of the `i` loop across the threads. In other words, approximately $n/\text{numthread}$ iterations are assigned to each thread in the team. Thread 0 will have iterations $i = 1, 2, \dots, n/\text{numthread}$, thread 1 will have $i = n/\text{numthread} + 1, \dots, 2*n/\text{numthread}$, and so on. Any remainder in $n/\text{numthread}$ is assigned to the team of threads via a mechanism determined by the OpenMP implementation.

Our example of a parallelized sparse matrix multiply where we determine the minimum, maximum, and average times of execution could show some measure of load-imbalance. Each row of the sparse matrix has a different number of elements. If the sparse matrix has a dense block banding a portion of the diagonal and mostly diagonal elements elsewhere there will be a larger “load” on the thread that computes the components from the dense block. Figure 19 shows the representation of such a matrix and how it would be split by using the default OpenMP scheduling mechanisms with three threads. With the “static” distribution of work among the team of three threads, a severe load imbalance will result. This problem can be mitigated in several ways. One way would be to apply a chunk size in the static distribution of

Fig. 19. A sparse matrix that is dense in one area. Using our sparse matrix vector algorithm on three threads, we would access the matrix as shown.

Fig. 20. A sparse matrix that is dense in one area. Using our sparse matrix vector algorithm with the appropriate chunk size on three threads we would access the matrix as shown. This is more load-balanced than the default distribution of iterations to the team of threads.

work equal to the size of the dense block divided by the number of threads. This would lead to the distribution of work shown in Figure 20. This can be accomplished by modifying the `PARALLEL DO` directive of Figure 18 to

```
!$OMP PARALLEL DO SCHEDULE(STATIC, (SIZE_OF_DENSE_BLOCK/numthreads))
```

where `SIZE_OF_DENSE_BLOCK` must be determined before the `do` loop construct in the parallel region. Determining this value is added overhead on the parallelization of the serial code.

At times, more explicit control may be necessary. The same kind of explicit control necessary in the equivalent message-passing implementation. The algorithm can be scheduled explicitly with similar constructs such as the number of threads and the thread identifier. This is another advantage of OpenMP; in addition to incremental parallelization, a programmer can take as much explicit control as is necessary for a given algorithm.

Figure 21 shows an explicit parallelization of the sparse matrix multiply. The OpenMP library function `omp_get_thread_num()` returns the thread identifier in the range from 0 ... the number of threads minus 1. Each thread starts with the iteration that matches a thread identifier, and the “parallel” loop now increments by the number of threads. There is no longer a need for the `PARALLEL DO` directive because of the explicit control! This interleaves each iteration in order to a different thread, so the issues of load balance are minimized.

The C/C++ version of the example in Figure 21 would be more complicated because the reduction clause operators available in C/C++ do not include `MIN` or `MAX` functionality. No intrinsic functions are available for use in the reduction clause.

3.4 Data Dependencies and False Sharing

In parallelizing algorithms, one has to ensure that every memory write operation is essentially independent of other memory operations from other threads in the team. If the programmer writes to a location in one thread and reads that same location in another thread of execution, a dependency exists. Since OpenMP provides no control over which thread executes, the programmer must deal with this dependency either by scoping the appropriate variables (private or shared) or introducing synchronization mechanisms to ensure that the dependency is met. Mitigating these data race conditions or dependencies is at the heart of shared-memory parallel programming, since data communication is through shared variables. Chandra et al. have a good, somewhat formal, discussion of the process of identifying and removing these dependencies [9].

The mechanisms for dealing with these data dependencies often require some restructuring of code. For example, it may be necessary to split a loop that computes multiple quantities. The “fissioned” loops can be run in parallel but the original construct cannot. In other situations new intermediate

```

!      compute yvec = Amat*xvec
...
!$OMP PARALLEL REGION PRIVATE(i,t,k,timestart,timeend,numthread,tid)
!$OMP+      SHARED(n,row_ptr,amat,xvec,col_ind,yvec)
!$OMP+      REDUCTION(MIN:timemin) REDUCTION(MAX:timemax)
!$OMP+      REDUCTION(+:timeave)
      timestart = omp_get_wtime()
      tid = omp_get_thread_num() ! get the thread identifier
      numthread = omp_get_num_threads()

      do i = (tid+1),n,numthread
        t = 0.0d0 ! inner product of row i with vector xvec
        do k=row_ptr(i), row_ptr(i+1)-1
          t = t + amat(k)*xvec(col_ind(k))
        enddo
        yvec(i) = t ! store result in yvec(i)
      enddo
      timeend = omp_get_wtime()
      timemin = timeend-timestart
      timemax = timeend-timestart
      timeave = (timeend-timestart)/numthread
!$OMP END PARALLEL REGION

```

Fig. 21. Parallel sparse matrix-multiply code fragment, in Fortran, that times the operation and reduces the minimum, maximum, and average times. The concurrency is explicitly controlled with the thread identifier and the number of threads.

quantities may need to be introduced. These will add additional memory requirements and the overhead of generating those intermediates.

Code restructuring will certainly involve tradeoffs that may affect performance and thus force a specific way of parallelizing the algorithm. One such performance issue is that, although there is no formal data dependency, there is a performance degradation because of the nature of the memory locations being accessed by the threads in the team. If independent threads are writing to memory locations in the same cache line, there is no true data dependency because each thread is writing to separate memory locations. Unfortunately, since these locations are in the same cache line, performance is degraded because each write forces the data to be flushed from the other processor cache. This cache thrashing is called “false sharing.”

Can “false sharing” really impact the performance of a parallel algorithm? Yes. In fact, the algorithm presented in Figure 21 will suffer from false sharing. The write to `yvec(i)` in the first iteration of each thread all have elements contiguous in memory; e.g., thread 0 and thread 1 will interact via the cache. As the algorithm proceeds the effect may decrease because of the varying size of the number of elements in each row of the matrix; each iteration will take a different time to execute. One way to mitigate this is to block the access

```

!      compute yvec = Amat*xvec
...
      blocksize = 5 ! the number of iterations each thread gets
      numblocks = n/blocksize ! number of blocks of iterations
! a remainder means extra block
      if (mod(n,blocksize).ne.0) numblocks=numblocks+1

!$OMP PARALLEL REGION PRIVATE(ii,i,ilo,ihi,t,k)
!$OMP+      PRIVATE(timestart,timeend,numthread,tid)
!$OMP+      SHARED(n,row_ptr,amat,xvec,col_ind,yvec)
!$OMP+      SHARED(blocksize,numblocks)
!$OMP+      REDUCTION(MIN:timemin) REDUCTION(MAX:timemax)
!$OMP+      REDUCTION(+:timeave)
      timestart = omp_get_wtime()
      tid = omp_get_thread_num() ! get the thread identifier
      numthread = omp_get_num_threads()

      do ii = (tid+1),numblocks,numthread
        ilo = (ii-1)*blocksize + 1 ! start of each block
        ihi = min((ilo+blocksize-1),n)
        do i = ilo,ihi
          t = 0.0d0 ! inner product of row i with vector xvec
          do k=row_ptr(i), row_ptr(i+1)-1
            t = t + amat(k)*xvec(col_ind(k))
          enddo
          yvec(i) = t ! store result in yvec(i)
        enddo
      enddo
      timeend = omp_get_wtime()
      timemin = timeend-timestart
      timemax = timeend-timestart
      timeave = (timeend-timestart)/numthread
!$OMP END PARALLEL REGION

```

Fig. 22. Parallel sparse matrix multiply code fragment, in Fortran, that times the operation and reduces the minimum, maximum, and average times. The concurrency is explicitly controlled with the thread identifier and the number of threads and appropriate blocking of the parallelized iterations to avoid false sharing.

to the iterations and thus the writes to `yvec(i)`. The block size simply has to be large enough to ensure that the writes to `yvec(i)` in each thread will not be in the same cache line. Figure 22 shows the blocked algorithm that will avoid false sharing using explicit control of the concurrency among the team of threads. This explicit blocking could be accomplished by modifying the `PARALLEL DO` directive of Figure 18 to

```
!$OMP PARALLEL DO SCHEDULE(STATIC,5)
```

Other mechanisms can be used to modify the way iterations are scheduled. They are explored in more detail in references [9, 4, 5].

As another example for analysis and parallelization, we examine the simple cache-optimized matrix multiply in Figure 23. Our examination of this code

```

do j = 1, CCOLS
  do k = 1, BROWS
    Btmp = B(k,j)
    do i = 1, CROWS
      C(i,j) = C(i,j) + A(i,k)*Btmp
    enddo
  enddo
enddo

```

Fig. 23. Partially cache-optimized matrix-multiply: serial code.

suggests that we should maximize the work in each thread with respect to the overhead of the OpenMP parallelization constructs. In particular, we should parallelize the outermost loop. A glance at the memory locations with write operations indicates that only $C(i,j)$, $Btmp$, i , j , and k are relevant. For effective parallelization, the loop index variables must be different for each thread, thus accessing only appropriate parts of the matrix. A and B have only read operations. Since all threads need to know the dimensions of the matrices, $CCOLS$, $CROWS$, and $BROWS$ need to be shared among team members. Since we are parallelizing over the j loop, each thread has a unique set of j values; and since $Btmp$ is a function of j , each thread should have a unique $Btmp$ (i.e., $Btmp$ should be private to each thread).

This loop structure can be parallelized in many ways. The most straightforward is to use the combined parallel work sharing DO constructs. The parallel code based on our analysis is shown in Figure 24. The data in Table

```

!$OMP PARALLEL DO PRIVATE(i,j,k,Btmp)
  do j = 1, CCOLS
    do k = 1, BROWS
      Btmp = B(k,j)
      do i = 1, CROWS
        C(i,j) = C(i,j) + A(i,k)*Btmp
      enddo
    enddo
  enddo
!$OMP END PARALLEL DO

```

Fig. 24. Partially cache-optimized matrix-multiply: parallel code.

4 shows the performance on a four-processor SMP system. The scalability

indicates some overhead. On four threads the efficiency ranges from 97.1% to 95.4% with increasing matrix sizes. The performance could be improved by further optimizing the cache with a blocking algorithm.

Table 4. Timings in seconds for multiple runs of the OpenMP parallelized matrix multiply code.

	Matrix Rank			
number of threads	500	1000	1500	2000
4	0.43	3.56	12.05	28.37
3	0.57	4.65	15.67	37.05
2	0.84	6.92	23.28	55.12
1	1.67	13.57	45.87	108.27

3.5 Future of OpenMP

We have described in this section a robust programming model for the development of applications using OpenMP on shared-memory systems. There are many ways to tackle a parallel algorithm, from the application of simple directives to essentially full control basing the execution on the thread identifiers available. At this point we have described both message passing with MPI and thread programming with OpenMP. Some applications use both, with mixed results [16, 42, 56]. Hybrid MPI/OpenMP applications are emerging in part due to the nature of how clusters are evolving with larger processor counts per node. Hybrid MPI/OpenMP software development presents several challenges. The programmer interested in this hybrid model should get a sound understanding of both programming models separately and then begin to merge them. The programmer interested in this should carefully understand the MPI-2 scope of thread policy set up in the initialization phase of MPI-2 codes. The real trick in merging these two programming models is getting the code to work in four different modes: serially, with just OpenMP, with just MPI, and with both MPI and OpenMP [37]. The hybrid code in any of these modes should generate correct results regardless of how many threads are used at the thread level or how data is distributed among multiple processes. Current hybrid applications have been developed with a subset of these four modes due to the complexity of the resultant application. Primarily MPI communications are done only in the master thread of execution. Hybrid applications is an advanced topic in programming models and more research is in progress addressing the issues involved.

Cluster-Based OpenMP

At a recent workshop, Intel described a new offering, Cluster OpenMP [36], that is in beta testing. The idea is to provide a runtime infrastructure that

allows the OpenMP programming model to run on clusters. Intel’s offering can serve as a reference implementation for this idea, but it is limited to Itanium clusters at the moment. Intel has added directives and library functions to make clear distinctions between private, shared, and “sharable” data (data that is among processes, i.e., on another cluster node). Cluster-based OpenMP is a current topic in the research community and should be monitored as the research efforts demonstrate the effectiveness [63, 38, 52].

The ultimate goal of these efforts is to have the runtime environment provide good performance on clusters for OpenMP; comparable performance to hybrid MPI and OpenMP is required. The programming syntax of the value-added standard would allow incremental parallelism that is often difficult with MPI code development. Many issues must be considered in this environment. Remote process invocation is an issue that will be of interest because the landscape of clusters and communication interconnects is vast.

Specifications 2.5 and 3.0

Currently the merged OpenMP 2.5 specification is completed and is available for public comment.¹⁰ A major change in the OpenMP 2.5 specification is the merger of the Fortran and C/C++ specifications into a single document [6]. The ARB is also resolving inconsistencies in the specifications, expanding the glossary, improving the examples, and resolving some of the more difficult issues with respect to the flush semantics and the persistence of threadprivate data across multiple parallel regions.

The 3.0 specification is on hold until the 2.5 merger is done, but several topics are under discussion to expand the applicability of OpenMP. These include task parallelism to handle while loops and recursion, automatic scoping of variables, interaction with other thread models (e.g., POSIX threads), more control or definition of the memory model for NUMA-style shared-memory systems, and expanded schedule types and reusable schedules. As an example of the importance of the last issue, the guided schedule gives an exponential decay of the chunk size of iterations for a loop construct. The ability to control or change the decay rate is useful for improved performance of some algorithms. The 3.0 specification will also address many of the issues of nested parallelism that is in several implementations now. One major issue that needs to be considered is error reporting to the application. What happens if no more threads/resources are available? Currently, most implementations simply serialize the construct. A code developer may want to switch algorithms based on the runtime environment.

3.6 Availability of OpenMP

Most vendors provide OpenMP compilers, and several open source implementations are available. The OpenMP Web site [64] provides more information

¹⁰See the <http://www.openmp.org> website.

regarding their availability and function. There are also pointers for open-source implementations.

4 Distributed Shared-Memory Programming Models

Distributed shared-memory (DSM) programming models use a physically distributed memory architecture with some aspect of shared-memory technology. DSM models are not as popular as message-passing or direct shared-memory models but have many of the complications of both.

The goal for DSM technology is to facilitate the use of aggregate system memory, the most costly component of most high-end systems. Stated differently, most DSM programming models want to provide shared-memory-like programming models for distributed-memory systems. This aspect of shared memory can be effected in hardware or software. The hardware mechanisms are those with the highest performance and cost. Software mechanisms range from those that are transparent to the user to those coded explicitly by the user. Since obvious latencies exist in the software stacks of these implementations, performance still depends on the skill of the programmers using these technologies. DSM models are not as popular as message passing or direct shared-memory models but have many of the complications of both.

Software DSMs fall basically into three categories:

- Transparent operating system technology
- Language-supported infrastructure
- Variable/array/object-based libraries

DSMs that are transparent to the user often use a virtual-memory system with kernel modifications to allow for inter-node page accesses. This approach makes the programming straightforward in function, but getting good performance requires understanding the locality of the data and the way data movement happens. These systems include technologies such as ThreadMarks [43], InterWeave [10], Munin [8], and Cashmere [19].

Language-based infrastructure includes specialty languages such as High Performance Fortran (see section 4.1) and one that is now getting vendor support, Unified Parallel C (see Section 5.1).

Data-specific DSM libraries have been those most used by the high-performance computing community. They include the popular SHMEM programming model available on the Cray T3D and T3E systems[3]. These DSMs require that the programmer identify variables or objects that are shared, unlike OpenMP where everything is shared by default. Operations that separate shared and local variables require programmer control of the consistency appropriate for the algorithm. Data movement is neither automatic nor transparent; it must be coded explicitly or understood via implicit data movement from library interfaces.

4.1 High Performance Fortran

High Performance Fortran (HPF) is a distributed-memory version of Fortran 90 that, like OpenMP, relies on the use of directives to describe the features that support parallel programming. Because HPF uses directives, most HPF programs may be compiled by any Fortran 90 compiler and run on a single processor. HPF was developed by an informal group and published as a standard [33] in much the same way as MPI. In fact, the MPI Forum followed the same procedures used by the HPF Forum.

HPF is not as widely available as MPI and OpenMP but is still in use. A slight extension of HPF is in use on the Earth Simulator; an application using that version of HPF achieved a performance of 14.9 Teraflops and was awarded a Gordon Bell prize in 2002 [69]. In this section, we will touch on a few of the features of HPF and give one example. More information and some examples may be found in [17]; the full HPF standard is also available [48].

One of the most important steps in implementing a parallel program is distributing the data across the processes. This step can often be burdensome and error prone. HPF provides several directives that allow the programmer to easily and efficiently describe many data distributions. The most important of these is the `distribute` directive. For example, to distribute an array across all processes in blocks, use

```
real a(100)
!HPF$ DISTRIBUTE(BLOCK) a
```

Note that the HPF directive is a comment because it begins with an exclamation point and will be ignored by Fortran 90 compilers that do not support HPF. HPF supports several styles of data decomposition, including `BLOCK` (contiguous groups of elements across processes) and `CYCLIC` (round-robin assignment of elements across processes). One of the most attractive features of HPF is that the programmer may change the data distribution by changing only the `DISTRIBUTE` directive; the HPF compiler takes care of all of the changes to the code that are required by a different distribution.

The other important directive for data decomposition is the `ALIGN` directive. This tells the HPF compiler to align one distributed array with another. This lets the programmer provide information about the relationship between the use of elements of different distributed arrays to the compiler, which can be used by the compiler to produce more efficient code.

HPF provides additional directives; for example, there is a way to specify that a variable is involved in a reduction operation. Figure 25 shows a simple matrix-matrix multiply example. Note that, unlike the MPI case, program declares the sizes of the arrays, not just the part that is on a particular process. The HPF compiler handles all of the details of the data decomposition, including determining the sizes of the local versions of the arrays. This example does not include any of the code that would normally be used to implement cache and register blocking; such changes are necessary to achieve high performance.

```

program matmult
  integer, parameter :: n=1000
  real a(n,n), b(n,n), c(n,n)
  !HPF$ DISTRIBUTE(BLOCK,BLOCK)::C
  !HPF$ ALIGN A(i,*) WITH C(i,*)
  !HPF$ ALIGN B(*,j) WITH C(*,j)
  !
  a = 1
  b = 2
  do i=1,n
    do j=1,n
      c(i,j) = dot_product(a(i,:),b(:,j))
    enddo
  enddo
  write (*,*) c
end

```

Fig. 25. Simple HPF matrix multiply program.

4.2 SHMEM

SHMEM exists in implementations from various computer and interconnect vendors[3, 45, 46, 47]. In addition, a public-domain version—a generalized portable SHMEM, or GP SHMEM [66, 65]—has been augmented for use on clusters. The SHMEM model is an asynchronous one-sided message-passing or data-passing model. SHMEM assumes that computations are performed in separate address spaces and that data is explicitly passed. The asynchronous one-sided model assumes that a process can read (“get”) data or write (“put”) data from or to another process’s address space without the active participation of the second process. These one-sided operations are now a component of MPI-2 [54], and we encourage programmers to use that functionality as opposed to SHMEM (see Section 2.3). It will, however, take time for the functionality to propagate through all the vendor-supported MPI implementations.

SHMEM relies on remotely accessible data objects that are symmetric. These are data objects that have a known relationship among the local and remote addresses, such as Fortran common blocks or variables with the SAVE attribute, data allocated with `shpalloc` in Fortran or `shmalloc` in C or C++. SHMEM has a robust set of collective routines based on a triplet of arguments: the starting processor, log of the stride, and the number of processors involved. This power-of-two stride was required for the hardware of the T3D and T3E systems, but it is not generally applicable to clusters. GP SHMEM augmented this behavior to include arbitrary stride counts. The collective routines operate on the same symmetric data objects in multiple processes; this a requirement is made to improve efficiency.

SHMEM can be thought of as a middle ground between message passing and a full DSM language. SHMEM supports other operations such as work-shared broadcast and reduction, barrier synchronization, and atomic memory operations. An atomic memory operation is an atomic read-and-update operation, such as a fetch-and-increment, on a remote or local data object. Full barriers, barriers on a subset of processes, and a locking mechanism are also provided. There are some problems with SHMEM in that there is a name-space explosion because the interface does not include the size of the object being passed. For example, five different broadcast calls are available in the T3E implementation, `shmem_broadcast`, `shmem_broadcast4`, `shmem_broadcast8`, `shmem_broadcast32`, and `shmem_broadcast64`. Moreover, there is no standard for SHMEM, so other vendor or open-source implementations are free to augment the library as their needs arise. This augmentation is often via environment variables where the default values may or may not provide optimal performance.

4.3 Global Arrays

The Global Arrays (GA) Toolkit [59, 60, 58] was designed to offer the best functionality of both distributed-memory and shared-memory programming models. In fact, GA requires the use of a message-passing library so an application can use message-passing algorithms in addition to the GA algorithms. The data is divided into local data and logically shared data that can be accessed only through the user interface layer of the GA package. GA assumes that the data representation is arrays of multiple dimensions. This provides a NUMA view of the aggregate memory of the system. The data locality must be managed explicitly by the programmer, with the knowledge that remote data access is slower than local data access. Figure 26 represents the view of the data structures in an GA application. The cost of remote data access promotes data reuse and locality of reference.

Fig. 26. View of data structures in Global Arrays.

The GA toolkit allows the user complete control over the data distribution to match any algorithmic needs. The user can have the library distribute the data automatically or can identify a specific dimension or block size for distribution. Complete irregular distributions are also possible. The locality information of data is also available. For example, a specific multidimensional patch of a GA that is required for an algorithmic computation may exist on one or more processes; the locality information is an array of the process identifiers.

Figure 27 shows the computational flow of a GA application. Data is extracted from “global” memory to “local” memory. The process then computes on that portion of the array copied to local memory. The results are copied or accumulated to global memory for further processing as the algorithm dictates.

Fig. 27. Computational flow of a GA application.

Copy operations from the “global” data to “local” data and the reverse are the fundamental functionality of GA. In addition, the locality information provided allows direct access to data “owned” by a given process. This arrangement allows for virtually any needed data parallel operations. Several built-in data-parallel-like operations are provided, including zeroing an array, filling an array with an arbitrary value, printing an array, and scaling an array by a constant.

GA has several language interfaces: C, C++, Fortran 77, Fortran 90/95, and Python. There is also a common component architecture (CCA) component version. In addition, the library provides language interoperability for

mixed-language applications. Arrays created and used in Fortran can be accessed by using the other language interfaces. Internal storage is, by default, that of the Fortran language but can be made either row or column major.

The library has evolved from the initial development for NWChem, a computational chemistry suite [73, 44], to meet requirements of new application areas. Ghost cells and sparse data structures were added to provide functionality for halo-like simulations and Grid-based codes, respectively. The data movement engine was separated from the original implementation and now provides a portable one-sided communication tool, the Aggregate Remote Memory Copy Interface (ARMCI). ARMCI handles the actual data transfers, synchronization operations, and memory management. GA also has a secondary storage mechanism, disk resident arrays (DRAs). DRAs extend the memory hierarchy one additional level; they allow for out-of-core algorithm development as well as internal checkpointing of data. Furthermore, GA offers interfaces to third-party libraries such as ScaLAPACK.

GA provides portable performance; it runs on most major cluster interconnect technologies and high-end supercomputers. ARMCI, the data movement engine, is tuned to the fastest mechanisms available on various platforms. The developers have strong interactions with vendor software and hardware engineers to keep the infrastructure current and the performance at the highest level. GA will continue to expand to meet the requirements of the user community as the need arises.

To give a flavor of GA programming, we present a simple blocked matrix-multiply routine in Figure 28. The function stores the product of two matrices A and B in the resultant C matrix. The assumption is that the GAs for each matrix are created and A and B are filled prior to calling the routine and that all matrices are two dimensional.

GA provides a robust set of functionality. The toolkit is essentially the standard programming model for electronic structure computational chemistry codes, where most of the manipulations are contractions of multidimensional tensors of various orders into lower-order tensors. GA is also used in image processing, financial security forecasting, computational biology, fluid dynamics, and other areas. GA does not offer the full incremental parallelism of OpenMP, but the functional code is straightforward to generate and then tune for performance. Rapid prototyping is possible once the initial infrastructure is built. More information is provided on the Global Arrays home page [25].

5 Future Programming Models

We present here a few examples of what we delineate as future programming languages or models, not because they are new ideas, but because they are just now moving from the research community to the vendor community. Other programming languages should be considered if one is willing to live

```

        subroutine ga_simplematmul(g_c, g_a, g_b)
            implicit double precision (a-h,o-z)
            ! include files from the GA suite
            #include "mafdecls.fh"
            #include "global.fh"
            c
                omitting declaration of variables
            parameter (blocksize = 32) ! arbitrary block size
            c
                ga_inquire(g_a,typea,rowsa,colsa) ! get matrix dimensions
                ga_inquire(g_b,typeb,rowsb,colsb)
                ga_inquire(g_c,typec,rowsc,colsc)
            ! check that types and dimensions match if not call ga_error
            call ga_zero(g_c) ! zero the result
            blocksi = rowsc/blocksize + 1
            blocksj = colsc/blocksize + 1
            blocksk = colsa/blocksize + 1
            ! somehow allocate local arrays loca[blocksize][blocksize],locb,locc
            ! get the number of processes
            nproc = ga_nnodes()
            ! atomically get the next task an ordered count 0, 1, ...
            !
                across all processes
            mtask = nexttask(nproc)
            itask = -1
            do ib = 1,blocksi
                ilo = (ib-1)*blocksize + 1
                ihi = min((ilo+blocksize-1),rowsc)
                mdg = ihi-ilo + 1
                itask = itask + 1
            ! parallelize over i blocks (ib variable)
                if (itask.eq.mtask) then
                    do kb = 1,blocksk
                        klo = (kb-1)*blocksize + 1
                        khi = min((klo+blocksize-1),colsa)
                        kdg = khi - klo + 1
            ! get patch of global A copied into local array loca
                        ga_get(g_a,ilo,ihi,klo,khi,loca,mdg)
                        do jb = 1,blocksj
                            jlo = (jb-1)*blocksize + 1
                            jhi = min((jlo+blocksize-1),colsc)
                            ndg = jhi - jlo + 1
            ! get patch of global B copied into local array locb
                            ga_get(g_b,klo,khi,jlo,jhi,locb,kdg)
            ! use optimize BLAS locally to compute patch of in locc
                            call dgemm('n','n',mdg,ndg,kdg,1.0d00,
                                +
                                    loca,mdg,locb,kdg,0.0d00,locc,mdg)
            ! accumulate into global array C from local locc
                            ga_acc(g_c,ilo,ihi,jlo,jhi,locc,mdg,1.0d00)
                        enddo
                    enddo
                    mtask = nexttask(nproc)
                endif
            enddo
        end
    
```

Fig. 28. Simple GA matrix multiply routine.

on the “bleeding edge” of technology—that is, with very robust features of the language and little support. Of particular note is Titanium [76], a high-performance Java dialect with extensions needed by scientific applications. We close this section with a view of what is next beyond near-term extrapolation of current technology and what is needed to really reach petaflops.

5.1 Unified Parallel C

Unified Parallel C [20] (UPC) is a parallel extension of the ANSI C standard. UPC, like Co-Array Fortran (see Section 5.2), has the advantage of extending a well-known and well-understood language for parallel computation. The development of the UPC language started with ANSI C and included experiences from various distributed parallel computing language efforts in the research community, with input from vendors, users, and academia.

UPC is a distributed shared-memory parallel programming language. The execution model assumes a number of threads working independently in a single-program multiple-data (SPMD) paradigm. The language provides synchronization when needed via barriers, the memory consistency model, and explicit locks. The memory in the language is logically split into private and shared memory with an affinity for a specific thread. Any thread can read from the globally shared address space, and the language extension includes identifying which data and pointers to data are “shared” among the threads. Figure 29 represents the memory layout in the UPC model.

Fig. 29. UPC memory model with respect to the thread affinity.

The SPMD nature of the model allows for work distribution based on the thread identifier, MYTHREAD, and the number of threads involved, THREADS. MYTHREAD and THREADS are keywords in the UPC language. The actual translation depends on the underlying runtime infrastructure, but that is transparent to the user from a functional point of view and is the responsibility of the compiler.

Because threads share memory and because portions of shared memory have affinity to specific threads, access to that memory has a sequencing issue that depends on the underlying runtime environment. Developing a UPC application simply requires specifying either a “strict” or a “relaxed” memory consistency mode. This specification can be done for the entire program, for a defined block of code, or for a specific variable or array. The “relaxed” consistency mode allows memory accesses in each thread to follow normal ANSI C models, ignoring access to “local” shared-memory references from other threads. When using the relaxed mode, the programmer is ultimately responsible for handling any synchronization necessary. The “strict” mode follows normal ANSI C models while considering accesses from all threads.¹¹ Locks are provided to ensure atomic access to critical sections of code and the associated memory locations. Figure 30 shows a “Hello World” program similar to the one presented in the OpenMP discussion.

```
#include <stdio.h>
#include "upc_relaxed.h"
int main(int argc, char *argv[])
{
    int tid;
    {
        tid = MYTHREAD;
        printf("<%d> of %d Threads\n",tid,THREADS);
    }
}
```

Fig. 30. “Hello World” UPC code,

The sharing of data is explicitly coded in the use of the “shared” qualifier or in how memory is allocated dynamically with the UPC memory allocation routines. Since data being shared has affinity to threads, the user needs to control how data is laid out. Both the static and the dynamic memory modes allow for this. By default, elements of data arrays are distributed by element in a round-robin fashion to the shared-memory region of each thread. This can easily be blocked to distribute rows or columns of matrices to each thread.

In addition to the SPMD use of the thread identifier and the number of threads to share work among threads, there is a work-sharing construct `upc_forall`. The functional form of this construct is similar to the standard for loop construct but with an extra affinity parameter:

```
upc_forall ( init-expr ; cond-expr; incr-expr; affinity),
```

where `init-expr`, `cond-expr`, and `incr-expr` are the ANSI C equivalent expressions. The affinity parameter can be either a variable or an address to

¹¹This is a simplification of the consistency model; consult the UPC specifications [20] for more details.

a variable. The affinity expression controls which thread actually computes an iteration of the loop construct. For a variable the thread that executes the loop is `MYTHREAD == variable%THREADS`. For an address the thread that executes the loop is `MYTHREAD == upc_threadof(address)`. The UPC library function `upc_threadof` identifies the thread that has affinity for the address argument.

The UPC language has great potential for providing long-term portability and performance for a wide variety of applications. We have provided only a taste of the language. The UPC Web site¹² provides many more details, examples, and availability of compilers.

5.2 The Co-Array Fortran Extension to Fortran 95

Co-Array Fortran [62] is an alternative parallel programming language based on an extension to Fortran 95. It uses a simple syntax that is intuitively natural to a Fortran programmer. It adopts a purely local view of data and computation, but it allows the programmer to make local data globally visible by declaring some variables to be co-arrays. A co-array is a Fortran 95 object, whether an intrinsic object or a user-defined derived type, that is declared with a co-dimension. For example, the declaration,

```
real :: x[*]
```

defines a scalar co-array object that is replicated across program images. The asterisk notation `[*]` indicates that program images are *virtual* images, replicated copies of a program within the SPMD programming model.

The actual number of images is determined when the program starts execution. The runtime system assigns images to physical processors in a platform-specific manner, for example, as processes or threads. The number of images is fixed; it may be the same as the number of physical processors, it may be greater, or it may be less. Each physical processor may be responsible for more than one image, for example, taking work from a task queue. Conversely, more than one physical processor may be responsible for the same image, for example, by spawning threads within a process to share the work. The programmer decides whether an image works only on its own local data or, using co-array syntax, works on data that it does not own, by making local copies of data owned by other images.

Co-dimensions may be multidimensional just like normal dimensions. Programmers can use them to represent a *logical* decomposition of virtual images that corresponds to a logical decomposition of a physical problem. For example, a two-dimensional field decomposed into blocks, as commonly used in weather, climate, and ocean codes, might be declared with two co-dimensions.

```
real :: field(m,n)[p,*] .
```

¹²See <http://upc.gwu.edu>.

In this case, each image holds a patch of the field of local size ($m \times n$). The asterisk notation indicates that the number of images is determined when the program starts execution, but the programmer wants to think of the images within a two-dimensional grid with p images in the first dimension.

For many applications that use finite difference operators to solve partial differential equations, for example, programmers often add halo cells around the local field data.

```
real :: field(0:m+1,0:n+1)[p,*]
```

The main communication requirement is the exchange of halo data, which, using Co-Array Fortran syntax, can be written with just a few lines of code [7, 61]. For example, the exchange in the east-west direction

```
field(1:m,0) = field(1:m,n)[p,q-1]
field(1:m,n+1) = field(1:m,1)[p,q+1]
```

can be written with two lines of code, where the programmer has adopted the convention that the first co-dimension represents the north-south direction and the second represents the east-west direction. The image corresponding to $[p,q]$ fills its lower halo with data from its west neighbor $[p,q-1]$ and its upper halo with data from its east neighbor $[p,q+1]$. Since co-array syntax allows an image to read or write data owned by any other image, it is the programmer's responsibility to provide appropriate synchronization.

Basing a parallel programming model on a simple extension to an existing language has a number of advantages. First, the programmer need not learn a new language. Co-array syntax is natural and familiar to the Fortran programmer. Second, the co-array extension can be implemented by using existing compiler technology. Co-dimensions behave, in most respects, like normal dimensions. Third, since the new parallel syntax becomes part of the language, the programmer can use it to write customized communication patterns that fit a particular problem, without being restricted solely to those patterns provided by a library. Fourth, the compiler can generate optimized code that takes advantage of specific features of specialized hardware on particular platforms. For example, in the halo exchange example, it can schedule communication to overlap with computation and to exercise multiple hardware channels simultaneously. Fifth, code written with Co-Array Fortran is portable [13]. Because the extension is part of the language, a compiler must implement it for all platforms it supports.

5.3 Beyond Future Programming Models

Programming language design follows system architecture development, at least in the domain of performance-critical computation, including high-performance computing. Language serves as the medium between a user's application and the underlying execution target platform. The challenge to

programming is to extract the best possible performance from the target parallel computer system for a given application while retaining correctness. The degree of difficulty (length of programming time) strongly depends on the ease of performance tuning.

Historically, a healthy tension dominating language design has existed between language abstraction to hide system complexity from the programmer and low-level language constructs to expose the system mechanisms for direct and precise control to achieve the best performance. However, parallel programming methods have been heavily oriented toward constructs providing explicit control of low-level mechanisms because the principal target architectures, including massively parallel processors (MPPs) and commodity clusters, provide little or no support for automatic management of system-wide parallel computation—hence the popularity of models such as MPI (e.g., MPICH-2) that expose the underlying system architecture in detail and give the programmer complete control of how the application program is mapped to the system resources, as well as the synchronization of their cooperative operation.

Unfortunately, current-generation high-end systems not only are difficult to program but often exhibit significant inefficiencies in operation, negating much of the advantage of exploiting existing commodity components. Future system architectures for high-end capability (as opposed to capacity) computing in the trans-petaflops performance regime may be custom designed for the purpose of global parallel execution, unlike conventional MPPs. While this assertion is considered controversial today, important projects are under way to achieve this (e.g., the DARPA HPCS program).

If real parallel computing systems reemerge, replacing (at least in part) aggregated ensembles of commodity microprocessors in the arena of high-end computing, programming methodologies and languages that represent them will be devised to reflect their new underlying architectures. While we cannot know in absolute terms what future programming languages will look like in this new petaflops computing world, it is possible to identify key attributes of such languages based on reasonable assumptions about such future machines. Examples of such assumptions include the following:

- Global address space such that any part of the system state can be accessed efficiently from any other execution site within the distributed system
- Relaxed consistency methods for efficient copy semantics
- Hardware support for efficient parallel execution for coarse-, medium-, and fine-grained parallelism
- Rapid context-switching with multi-threaded execution
- Automatic hardware-supported latency hiding
- Efficient synchronization for many forms of coordination including message passing, producer-consumer, message-driven, and object-oriented
- Dynamic adaptive resource management and load balancing
- Streaming processing for high-temporal-locality computing
- In-memory processing for high-bandwidth, low-locality computing

- High-global-bandwidth, low-latency system-wide communication

Future programming languages for custom petaflops-scale system architectures incorporating some or all of these properties will differ from conventional programming practices by providing constructs that support a richer descriptive semantics of application parallelism and locality, rather than imperative specification of explicit mapping of data and code to hardware elements as is done today. Latency will be hidden in such future machines by a variety of automatic methods, and a much wider range of forms of parallelism will be efficiently supported. Thus, the key challenge to future programming is to make available to compilers, runtime systems, and hardware architecture descriptions of algorithmic/application parallelism and the synchronization relationships among coordinated computing actions.

A secondary feature of such future languages is the ability to represent locality relationships of data and tasks at various levels of granularity as a source of hints or heuristics for assisting and guiding the system in allocating and assigning physical resources. This is very different from the conventional practice of the programmer asserting the exact resource allocation mapping. Not only does this advanced approach simplify programming, but it also allows the system to exploit runtime information in conjunction with programmer and compile-time information to determine optimal placement of logical objects on the distributed physical resources.

While a rich set of semantics for parallelism representation and locality relationship description may constitute a major part of future programming languages for custom-scalable petaflops-scale system architectures, additional language capabilities will be incorporated to deal with practical aspects of very large systems. Three factors in particular will drive innovation in future language design:

1. *Performance monitoring* will become an integral part of the compiler and language, not just to show the programmer the bottlenecks, but to permit advanced compilation and runtime systems to make direct use of observed operation characteristics for automatic performance tuning, with some guidance by the programmer.
2. *Microcheckpointing* will be used to identify key locations in the by the programmer. Microcheckpointing identifies key locations in the execution trace where subsets of total program data may be temporarily archived until some follow on release point is correctly accomplished, at which point the snapshot of the partial state may be garbage collected. These minor fall-back points are employed when an error is detected in subsequent execution without having to restart the entire program.
3. *Advanced input/output constructs* will be used for generating “information products.” It will become increasingly impractical to attempt to store the full raw data from a simulation because the data sets will become prohibitively large. Also, the data itself, even if visualized, may not be useful in understanding the implications and consequences of the results.

An output layer to process the raw data may be necessary to generate information products that can be many orders of magnitude smaller than the basic data values but far more meaningful to the scientist or engineer. Future languages will emphasize high-level information rather than raw data sets as the principal output content, and the I/O semantics of the language will reflect this new usage.

6 Final Thoughts

The information in this chapter touches only the tip of the iceberg with respect to the issues of writing parallel programs. Even long-term practitioners fall into the many pitfalls of developing parallel codes. Overall, writing parallel programs is best learned by “getting your hands dirty.”

It is important to use the technology needed to get the job done, but it is also important to think about what changes might come in the future. The software development research community is producing new technologies rapidly and some of these technologies may be useful in high-performance application development. Although implementing object-oriented technology in Fortran 77 is impossible, some of the object-oriented concepts can build better-structured Fortran 77 codes. For example, abstraction and data-hiding are easily implemented with solid APIs for the functionality required.

What will come in the future? In this book, the chapter on common component architecture technology [1] discusses how the CCA framework has been used successfully to integrate functionality among multiple computational chemistry codes on parallel platforms. Also, the cross-cutting technologies of aspect-oriented programming [21] could change the way in which we construct software infrastructure for event logging, performance monitoring, or computational steering.

One additional comment is in order. Readers new to parallel computing on clusters might ask which is the best programming model with respect to performance and scalability. These are only two aspects of the interaction with a programming model and an application code with many different algorithms. The programming model also determines the ease of algorithmic development and thus application development and maintenance. Asking which is best is similar to asking which preconditioner, which Kyrlov subspace method, or which editor is the best to use. All programming models have strengths and weaknesses, and the choice is best made by those actually using the programming model for their particular purpose. MPI offers the greatest availability, portability, and scalability to large systems. OpenMP offers very good portability and availability with reasonable scalability on SMP systems. The distributed shared-memory programming models are best when long-term availability is possible and there is an appropriate match to the algorithms or applications involved. Clearly, programming models and their associated execution models will have to evolve to be able to reach sustained petaflops levels

of computing, which will in time move to computational resources known as clusters.

Finally, we will put together a series of examples of “working” code for many of the programming models discussed in this chapter. These will be designed around small computational kernels or simple applications in order to illustrate each model. The examples will be available at the Center for Programming Models for Scalable Parallel Computing website¹³.

7 Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38 with Argonne National Laboratory and under Contract W-7405-ENG-82 at Ames Laboratory. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, non-exclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. We thank the members of the Center for Programming Models for Scalable Parallel Computing [67] who have helped us better understand many of the issues of parallel software development and the associated programming models. We thank Brent Gorda, Angie Kendall, Gail W. Pieper, Douglas Fuller, and Professor Gary T. Leavens for reviewing the manuscript. We also thank the book series editors and referees for their many helpful comments.

References

1. Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott R. Kohn, Lois McInnes, Steve R. Parker, and Brent A. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 8th High Performance Distributed Computing (HPDC'99)*, 1999. <http://www.cca-forum.org>.
2. S. Balay, W. D. Gropp, L. Curfman McInnes, and B. F. Smith. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.0, Argonne National Laboratory, 2001.
3. R. Bariuso and A. Knies. SHMEM's user's guide, 1994. Eagan, MN, Cray Research, Inc. SN-2515 Rev. 2.2.
4. OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface, Version 2.0*. OpenMP Architecture Review Board, November 2000. <http://www.openmp.org/drupal/mp-documents/fspec20.pdf>.

¹³At this URL: <http://www.pmodels.org/ppde>

5. OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface, Version 2.0*. OpenMP Architecture Review Board, March 2002. <http://www.openmp.org/drupal/mp-documents/cspec20.pdf>.
6. Mark Bull. OpenMP 2.5 and 3.0. In *Proceedings of the Workshop on OpenMP Applications and Tools, WOMPAT 2004*, Houston, TX, May 17-18 2004. Invited Talk.
7. Paul M. Burton, Bob Carruthers, Gregory S. Fischer, Brian H. Johnson, and Robert W. Numrich. Converting the halo-update subroutine in the MET Office unified model to Co-Array Fortran. In Walter Zwiefelhofer and Norbert Kreitz, editors, *Developments in Teracomputing: Proceedings of the Ninth ECMWF Workshop on the Use of High Performance Computing in Meteorology*, pages 177–188. World Scientific Publishing, 2001.
8. J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, 1991.
9. Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.
10. DeQing Chen, Sandhya Dwarkadas, Srinivasan Parthasarathy, Eduardo Pinheiro, and Michael L. Scott. Interweave: A middleware system for distributed shared state. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 207–220, 2000.
11. E. Chow, A. Cleary, and R. Falgout. HYPRE User’s manual, version 1.6.0. Technical Report UCRL-MA-137155, Lawrence Livermore National Laboratory, Livermore, CA, 1998.
12. D. Clark. OpenMP: A parallel standard for the masses. *IEEE Concurrency*, 6(1):10–12, January–March 1998.
13. Cristian Coarfa, Yuri Dotsenko, Jason Lee Eckhardt, and John Mellor-Crummey. Co-array Fortran performance and potential: An NPB experimental study. In *The 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, College Station, Texas, October 2003.
14. Cray Research. *Application Programmer’s Library Reference Manual*, 2nd edition, November 1995. Publication SR-2165.
15. Leonardo Dagum and Ramesh Menon. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January–March 1998.
16. Suchuan Dong and George Em. Karniadakis. Dual-level parallelism for deterministic and stochastic CFD problems. In *Proceedings of Supercomputing, SC02*, Baltimore, MD, 2002.
17. Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors. *Sourcebook of Parallel Computing*. Morgan Kaufmann, 2003.
18. Paul F. Dubois. Ten Good Practices In Scientific Programming. *Computing in Science & Engineering*, 1(1), January-February 1999.
19. Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 153–159, Los Alamitos, CA, April 1999. IEEE Computer Society.

20. Tarek A. El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC Language Specifications Version 1.1.1, October 2003. http://www.gwu.edu/upc/docs/upc_spec.1.1.1.pdf.
21. Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-Oriented Programming. *Communications of the ACM*, 44(10):29–32, October 2001.
22. Earth Simulator home page, <http://www.es.jamstec.go.jp>.
23. Mike Folk, Albert Cheng, and Kim Yates. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of Supercomputing'99 (CD-ROM)*, Portland, OR, November 1999. ACM SIGARCH and IEEE.
24. Parallel Computing Forum. PCF Parallel FORTRAN Extensions. *FORTRAN Forum*, 10(3), September 1991. (special issue).
25. Global Array Project. <http://www.emsl.pnl.gov/docs/global>.
26. William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, 1998.
27. William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd edition. MIT Press, Cambridge, MA, 1999.
28. William Gropp, Ewing Lusk, and Thomas Sterling, editors. *Beowulf Cluster Computing with Linux*. MIT Press, 2nd edition, 2003.
29. William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.
30. William D. Gropp. Learning from the success of MPI. In Burkhard Monien, Viktor K. Prasanna, and Sriram Vajapeyam, editors, *High Performance Computing – HiPC 2001*, number 2228 in Lecture Notes in Computer Science, pages 81–92. Springer, December 2001. 8th International Conference.
31. The Open Group. *System Interfaces and Headers, Issue 4, Version 2*. The Open Group, 1992. <http://www.opengroup.org/public/pubs/catalog/c435.htm>.
32. R. Hempel and D. W. Walker. The emergence of the MPI message passing standard for parallel computing. *Computer Standards and Interfaces*, 21(1):51–62, 1999.
33. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1–2):1–170, 1993.
34. J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPLib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, December 1998.
35. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
36. Jay Hoeftinger. Towards industry adoption of OpenMP. In *Proceedings of the Workshop on OpenMP Applications and Tools, WOMPAT 2004*, Houston, TX, May 17–18 2004. Invited Talk.
37. Forrest Hoffman. Writing hybrid MPI/OpenMP code. *Linux Magazine*, 6(4):44–48, April 2004. <http://www.linux-mag.com/2004-04/extreme.01.html>.
38. Y. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. OpenMP for networks of SMPs. In *Proceedings of the 13th International Parallel Processing Symposium*, April 1999.
39. Paul Hyde. *Java Thread Programming*. SAMS, 1999.
40. IEEE, editor. *IEEE Standard for Information Technology-Portable Operating System Interface (POSIX)*. IEEE Standard No.: 1003.1, 2004.

41. Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, William Gropp, and Rajeev Thakur. High performance MPI-2 one-sided communication over InfiniBand. Technical Report ANL/MCS-P1119-0104, Mathematics and Computer Science Division, Argonne National Laboratory, 2004.
42. Gabriele Jost, Jesus Labarta, and Judit Gimenez. What multilevel parallel programs do when you are not watching: A performance analysis case study comparing MPI/OpenMP, MLP, and nested OpenMP. In *Proceedings of the Workshop on OpenMP Applications and Tools, WOMPAT 2004*, pages 29–40, Houston, TX, May 17–18 2004. Invited Talk.
43. Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Tread-Marks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, January 1994.
44. R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, and A. T. Wong. High performance computational chemistry; an overview of NWChem a distributed parallel application. *Computer Physics Communications*, 128:260–283, 2002.
45. Alphaserwer SC user guide, 2000. Bristol, Quadrics Supercomputer World Ltd.
46. Scali library user’s guide, 2002. Oslo, Norway, Scali.
47. Message Passing Toolkit: MPI programmer’s manual, document number : 007-3687-010, 2003. Mountain View, CA, Silicon Graphics Inc.
48. C. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, and M. E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.
49. B. Leasure, editor. *PCF Fortran: Language Definitions, Version 3.1*. The Parallel Computing Forum, Champaign, IL, 1990.
50. J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of SC2003*, November 2003.
51. Z. Li, Y. Saad, and M. Sosonkina. pARMS: A parallel version of the algebraic recursive multilevel solver. *Numerical Linear Algebra with Applications*, 10:485–509, 2003.
52. Ricky Kendall Lie Huang, Barbara Chapman. OpenMP on distributed memory via global arrays. In *Proceedings of Parallel Computing 2003 (ParCo2003)*, Dresden, Germany, September 2–5 2003.
53. Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.
54. Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998.
55. Papers about MPI. <http://www.mcs.anl.gov/mpi/papers>.
56. Kengo Nakajima and Hiroshi Okuda. Parallel Iterative Solvers for Unstructured Grids Using and OpenMP/MPI Hybrid Programming Model for GeoFEM Platform on SMP Cluster Architectures. *Lecture Notes in Computer Science*, 2327:437–448, 2002.
57. Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrel. *Pthreads Programming*. O’Reilly & Associates, Inc, 1996.
58. J. Nieplocha, R. Harrison, M. Krishnan, B. Palmer, , and V. Tipparaju. Combining shared and distributed memory models: Evolution and recent advance-

- ments of the Global Array Toolkit. In *Proceedings of POOHL'2002 workshop of ICS-2002*, New York, NY, 2002.
59. Jarek Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A portable “shared memory” programming model for distributed memory computers. In *Proceedings of Supercomputing 1994, SC94*, pages 340–349, 1994.
 60. Jarek Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:197–220, 1996.
 61. Robert W. Numrich, John Reid, and Kieun Kim. Writing a multigrid solver using Co-Array Fortran. In Bo Kågström, Jack Dongarra, Erik Elmroth, and Jerzy Waśniewski, editors, *Applied Parallel Computing: Large Scale Scientific and Industrial Problems*, 4th International Workshop, PARA98, pages 390–399, Umeå, Sweden, June 1998. Springer. Lecture Notes in Computer Science 1541.
 62. Robert W. Numrich and John K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998.
 63. The Cluster Enabled Omni OpenMP Compiler, <http://phase.hpcc.jp/Omni/Omni-doc/omni-scash.html>.
 64. OpenMP ARB home page, <http://www.openmp.org>.
 65. K. Parzysek and R. A. Kendall. GPSHMEM: Application to kernel benchmarks. In *Proceedings of the Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 404–409, Cambridge, MA, November 4–6 2002. ACTA Press, Anaheim, CA.
 66. K. Parzysek, J. Nieplocha, and R. A. Kendall. A generalized portable SHMEM library for high performance computing. In M. Guizani and X. Shen, editors, *Proceedings of the IASTED Parallel and Distributed Computing and Systems 2000*, pages 401–406, Las Vegas, Nevada, November 2000. IASTED, Calgary.
 67. Center for Programming Models for Scalable Parallel Computing. <http://www.pmodels.org>.
 68. Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, NASA Ames Research Center, Moffett Field, CA, 1990.
 69. Hitoshi Sakagami, Hitoshi Murai, Yoshiki Seo, and Mitsuo Yokokawa. 14.9 TFLOPS three-dimensional fluid simulation for fusion science with HPF on the Earth Simulator. In *Proceedings of Supercomputing*, 2002.
 70. C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, January 1985.
 71. B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multi-level Methods for Elliptic Partial Differential Equations*. Cambridge University Press, New York, 1996.
 72. Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1995.
 73. T.P. Straatsma, E. Aprà, T.L. Windus, W. E.J. de Jong E. J. Bylaska, S. Hirata, M. Valiev, M. T. Hackler, L. L. Pollack, R. J. Harrison, M. Dupuis, D.M.A. Smith, J. Nieplocha, V. Tipparaju, M. Krishnan, A. A. Auer, E. Brown, G. Cisneros, G. I. Fann, H. Fruchtl, J. Garza, K. Hirao, R. A. Kendall, J. Nichols, K. Tsemekhman, K. Wolinski, J. Anchell, D. Bernholdt, P. Borowski, T. Clark, D. Clerc, H. Dachsel, M. Deegan, K K. Dyll, D. Elwood, E. Glendening, M. Gutowski, A. Hess, J. Jaffe, B. Johnson, J. Ju, R. Kobayashi, R. Kutteh, Z. Lin, R. Littlefield, X. Long, B. Meng, T. Nakajima, S. Niu, M. Rosing, G. Sandrone, M. Stave, H. Taylor, G. Thomas, J. van Lenthe, A. Wong, and Z. Zhang.

- NWChem, A computational chemistry package for parallel computers, Version 4.6, 2004. Pacific Northwest National Laboratory, Richland, WA.
74. Rajeev Thakur, William Gropp, and Brian Toonen. Minimizing synchronization overhead in the implementation of MPI one-sided communication. In Dieter Kranzlmüller, Peter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, pages 57–67. Springer Verlag, 2004. 11th European PVM/MPI User’s Group Meeting, Budapest, Hungary.
 75. ANSI X3H5. *FORTRAN 77 Binding of X3H5 Model for Parallel Programming Constructs*. Draft Version, 1992.
 76. Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice And Experience*, 10(11–13):825–836, 1998.